The words played all around our heads
Perhaps we went too far
We'll soldier on until the end again

—We Stand Alone, Ultravox [Rage in Eden]

Created on the Apple Macintosh Plus with Microsoft Word and Claris MacDraw II
Printed in 10-point Times Roman, diagrams in Avant Garde and source code in Courier

Printing date 2000/10/02

Preface

This report describes the work towards the fulfillment of the 250-hour assignment course with the Integrated-Circuit and Electronics group of the University of Twente, under the supervision of Eddie van Schie and Philip Wolbert, executed during March through June 1990.

The assignment consists of the design and implementation of Phase II of *TRIP*, the TRENDY Input Processor. TRENDY is an integrated-circuit fabrication process simulator which takes a textual input file to specify the steps of the process to be simulated. It is the purpose of the TRIP software to allow users of TRENDY to instead employ interactive graphical manipulation techniques to specify these steps rather than typing the actual commands. TRIP will then generate these from the graphical representation.

Actually, a very substantial amount of effort was expended towards the consideration that TRIP could also be used as an input generator for other text-based applications. Experience has shown [6] that there could certainly be a great use for such a TRIP system for SUPREM 3 simulations. In this sense the name 'TRIP' is a little bit of a misnomer and something like 'XIP' might have been more indicative of the nature of the program. The discussion in this report, however, generally refers to just the TRENDY simulator.

Phase I of the implementation, described in *Design and Partial Implementation of TRIP* [8] covered the design of the TRIP system and the version 1.0 implementations of the Command Manager and Library Manager. These allow the specification of a particular text-based application's command set in a designated file, called the *Command Dialog Description file,* and manage a command library system.

This document describes the Phase II work of the TRIP implementation—specifically, versions 1.0 of the Dialog Manager, Editor Manager and the Layout Manager, as well as the version 1.1 updates to the Command Manager. It is responsible for allowing users of TRIP to interactively set and modify parameters to simulator commands. It does this through the use of *dialogs*, which display the command parameters to a user in an attractive and intuitive manner. The problem of dialog generation is made more complex than otherwise because the CDD specifications of the command set are not known until run-time. It is by changing these specifications that dialogs can be generated for different applications.

The final Phase III should complete TRIP with the implementation of what is at least at this point in time known as the Structure Manager. This would let users manage the interrelationships between individual commands, as well as generate the actual input for the simulator.

This report

Because the software relies so heavily on it, the reader is assumed to be familiar with the basics of the X window system and the Xr toolkit. This includes concepts such as window exposures, event processing and field editors, although the innovative new aspects of X are explained briefly in the text.

Especially because this work is part of a larger and still evolving effort, I have tried to pay particular attention to the creation of a coherent presentation of the way in which the software itself works. This implies rather extensive coverage of the procedural interfaces of each of the new managers. I believe it is essential that software created in a cooperative environment be well-documented, or future programmers may not completely understand its concepts and run the risk of creating inefficient solutions.

The C source code fragments in the text use the syntax of the fairly recently approved ANSI C standard described in [5], even though the actual code is written for an old-style compiler. I feel this is justified by the enhanced expressiveness of the new standard. For example, in the new standard function prototypes allow explicit specification of the parameter type.

Often at the end of a section or chapter, additional material may be found which will be of particular interest to programmers who wish to understand details of the internal operation of the implementation, perhaps in order to modify or enhance the functionality already provided. These sections, which may be skipped by readers and programmers who wish only to use the routines and managers as is, are marked by a vertical bar in the left margin, as this paragraph is.

Because not all the details of the implementation described in this document could be foreseen at the time of the original definition there are some slight changes to the terminology used in it. In particular, the scope of the Editor Manager has now been modified—any reference in the original report to "Editor Manager" should be considered to be replaced either by the new "Dialog Manager", or by all three of the new managers mentioned above collectively. In this report, "Editor Manager" refers only to a single module of TRIP which is used by the Dialog Manager.

The software

In writing the software I have made great pains to attempt to ensure that it was fault-tolerant. In this I mean that it should always suspect that at some point something may go wrong, and to program defensively so as to avoid unrecoverable situations. Of course, some errors are not directly under the program's control and thus are unavoidable—for instance, when a request for a system resource is denied. However, it would be intolerable for the user to have the entire program terminate just because a memory allocation request could not be fulfilled. In cases like this, it is most always possible to abort the single operation which resulted in the request (taking care, and this is also important, to release any resources obtained so far), and to allow the user to try again when conditions may be more favorable.

On the question of errors—although there is no problem in their detection and propagation to different parts of the software, signaling errors to the user has not yet been implemented. I would have liked to have introduced a scheme I have already used experimentally, which provides impatient users with concise and easy-to-understand error messages, and inquisitive users with detailed level-by-level explanations of what went wrong. The design of such a system was exhibited in [12]. Unfortunately, there was not sufficient time for me to implement this as well.

Although I believe the finished software performs as specified, some of the advanced features could not be completed in time for their incorporation in this report. Due to lack of time it was necessary to give precedence to the writing of the report. Nevertheless, I will attempt to have completed these by the time of the Phase III implementation. Currently, the finished Phase II software comprises some 4,500 lines of C code. The Phase I and II software together constitute about 9,000 lines.

Ben Hekster
Tankelanden 5
7542 DR Enschede
053-764091

Contents

1 Introduction

Ride the arrow to the target
Shake it hip to hip, rocket through the wilderness
Around the world, the trip begins with a kiss

—Roam, B-52s [Cosmic Thing]

TRIP makes extensive use of version 11 of the X Window system (commonly referred to simply as X11) as well as the Xr ("X-ray") toolkit for its graphics capabilities. [1-4] Using a portable window system as well as a very portable language (C) and operating system (UNIX) should allow the software to run on other systems with no, or at most very minor, modification to the program source.

Although TRIP was originally mandated to run with OSF/Motif, it should in fact run with or without any window manager at all. During development TRIP ran with the Hewlett-Packard Window Manager (*hpwm*), which is actually very similar to the Motif Window Manager (*mwm*) [11]. Information on using and customizing the X Window environment may be found in [10].

All the Phase II software written and described here is contained in the /users/eddie/ben directory. The Phase I files are contained in /users/philip/peter/trip.f. It was not possible to create links to the latter files, some of which were needed for testing purposes, because the two directories reside on different file systems. For reasons of efficiency and because of the problems involved in keeping two versions of the same software consistent, I thought it unwise to make actual copies of these files. Therefore, where they are needed, I used the full path name of the Phase I files. Depending on which file system is used for the final development of TRIP, these will have to be changed.

It is interesting to note that whereas many programs and software packages have one external interface, the Phase II TRIP software has three. First, there is the interface to the rest of the TRIP software, specifically the Phase I Command Manager and the as yet nonexistent Phase III Structure Manager. Then there are the external editors, where both sides of the communication were developed from scratch in this work. Lastly, there is the interface with the TRIP user himself. Some of the complexity of this software is a result of the fact that two of these entities, the user and the external editors, operate asynchronously from the main Phase II process.

OVERVIEW

The TRIP system, as it now stands, may be depicted in terms of its modules as follows:

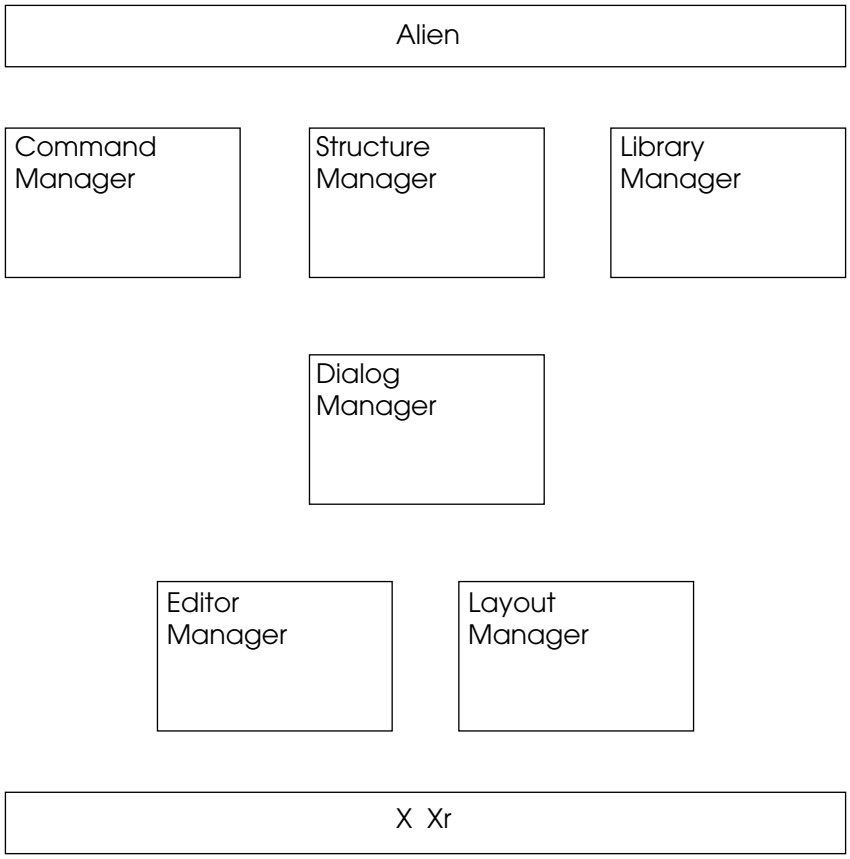

Figure 1-1. Overview of the TRIP system

The ‘alien’ module shown above is not actually part of the TRIP system proper—it is used for validation and demonstrational purposes but will not be a part of the final TRIP software. It derives its name from the parasitic creature in the science-fiction classic *Alien*, in which the organism, like the software module, latches tightly onto the host's external ‘interface.’ Specifically, it serves to fill the voids between the Phase I and II managers and to form a temporary bridge to the UNIX command-line interface. In addition, it composes valuable example code for the benefit of the implementor of the Phase III Structure Manager, which should help serve to reduce the considerable learning curve involved with learning to use a new complex window system and toolkit and software modules written by other programmers.

Changes from the original design structure proposed in [8] arose because the Phase II implementation described here brought about more work than originally expected, and thereby saw the introduction of two additional managers. The subsequent implementation of the Phase III software will unquestionably also introduce further changes to the architecture of the system.

The following figure shows an overview of the software that was implemented during this Phase II of the project. The outer boxes (with the exception of the ‘external editors’) represent conceptual ‘Managers,’ the inner boxes compilation units (source files). The Managers have been written to be highly independent of one another in the sense that intermanagerial communication proceeds only through well-defined entry points and not by random access in each other's private data structures. This means that any changes in the implementation of one Manager which still adhere to its specifications should not affect any other. The external editors also exhibit this independence but exist in separate compiled programs (hence the term *external*) which may be used by the TRIP system.

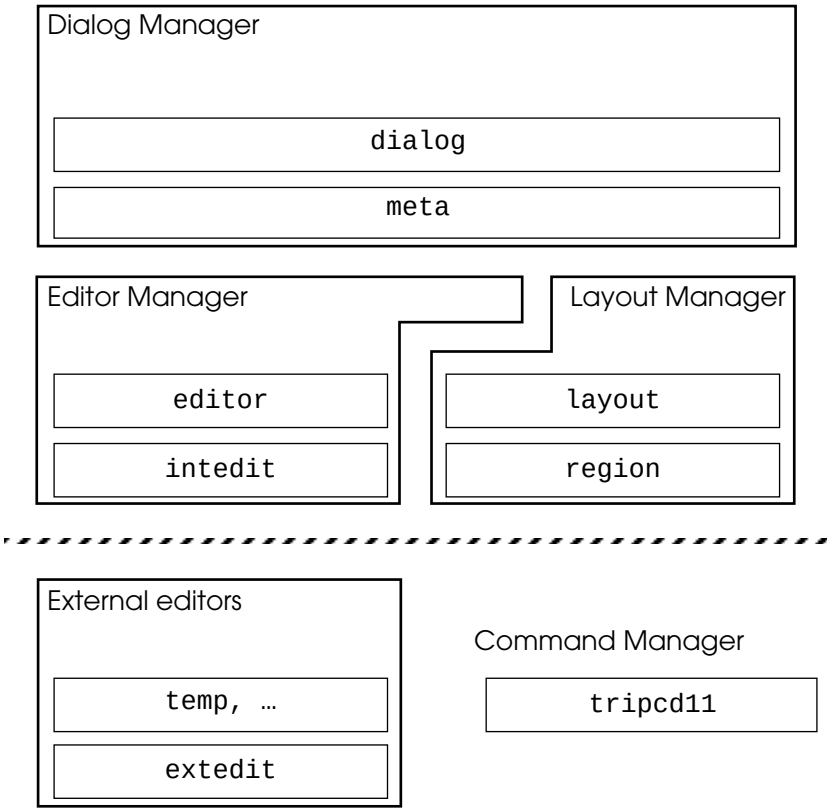

Figure 1-2. Overview of the new managers

Each manager has a well-defined function in terms of the rest of the system:

Dialog Manager. The Dialog Manager forms the interface of the system shown in figure 1-2 to the rest of the TRIP system—in particular, the Command Manager and the yet to be implemented Structure Manager. It provides the Structure Manager with a very easy-to-use interaction model in the form of *dialogs*. These are special windows in which a user can view and edit parameters of particular simulator commands. The dialog is an extremely convenient vehicle for the specification of simulation commands, because, unlike the original textual commands it does not require the user to memorize any command syntax, the number or types of command parameters, or the legal options for each parameter. The dialog implicitly (by limiting the selection of parameter values to those actually available) or explicitly (by warning the user that a specified value is not legal) excludes the possibility of syntax errors in the specification of the simulator input file. Also, the responsive graphical interface of the dialog should greatly enhance operator productivity. (Studies to quantify this claim are planned in the context of another course.) Simply, when the Structure Manager determines that the user wants to add a new or edit an existing command, it may simply instruct the Dialog Manager to open a dialog which allows the user to do so.

Editor Manager. A dialog consists of individual editors representing each of the parameters in a simulator command. The Editor Manager provides five basic types of editors, called *internal editors*: integer and real number, string, Boolean, and list editors. In special cases, which will depend on the particular type of output file TRIP is being used to generate, it may be desirable to write more effective special-purpose editors to replace an internal editor. Therefore, the Editor Manager supports the use of independently written *external editors* which reside in separate source files. External editors may be written to override an internal editor for a single or an entire class of parameters, or even for the handling of an entire command consisting of many parameters. The efficient management of the additional processes required is handled entirely transparently by the Editor Manager.

Layout Manager. After having translated a command's parameters into a number of individual editors, they must be placed together in the dialog window. The Layout Manager takes care of this very efficiently, while maintaining a pleasing layout of editors in the dialog. It allows for the hierarchical structuring of editors to ensure that related parameters are grouped together within the dialog. To cope with the varying sizes and shapes of single and groups of editors, the Layout Manager entertains the con-

cept of an arbitrary two-dimensional shape called a *region*, and contains some powerful routines for creating and manipulating them.

Command Manager. The tripcd11 files contain the version 1.1 enhancements to the Command Manager. This module reads the simulator input command and parameter specifications from the CDD file and builds a corresponding internal representation. It maintains the structuring of commands into parameters, and of parameters into data types. It also remembers which commands and data types have external editors associated with them.

Each of these managers is discussed separately in the following chapters. The programmer implementing the completing Phase III of TRIP should probably need to concern himself only with the Dialog and Command Manager information in this document. Programmers writing external editors need only the information in the Editor Manager chapter. Eventual users of TRIP who want to write or modify the CDD file will also find valuable information on the syntax of parameter specifications in the Editor Manager chapter.

Remarks

It is painfully obvious that the current hardware configuration of the ICE group's Hewlett-Packard computer system is inadequate to run the proposed software. Although the TRIP program itself would seem to make only very modest demands, the cumulative claims on the resources of the system by both the TRENDY simulator and the X window system are simply too great. Many a time I was not able to start X at all, or when I could, to compile and link a moderately-sized file or do some printing because of insufficient swap space, too many running processes or a full file system. However, running the software under these circumstances has made me more acutely aware of the need for efficient and defensive programming, and the program is probably the better for it.

Although the documentation [13] mentions it, the compiler on our system does not recognize the new ANSI C specification compilation mode (-Aa). This is a shame, because the ANSI specification offers much improved error-detection over the original definition.

OTHER CONTRIBUTIONS

Various smaller products were written for supporting the development of this software.

Shell scripts

Because I was not working under my own account I wrote a personalized shell script, *xbs*, which replaces the existing script xs. Besides defining some personally used environment variables, it changes the $HOME directory to ‘ben’ before starting the X Window system. This is important, because it allows another .Xdefaults file to be used than the one in the original home directory /users/eddie. This new X defaults file could be used for experimentation without disturbing the original account owner's settings. It is likely that the Phase III implementation will include the definition of some .Xdefaults variables which can be set by users to change window colors and other program attributes.

Another shell script, *bake* (for *b*eep m*ake*), is built on top of the UNIX make command. Although syntactically equivalent, it automatically invokes make as a background process, suppresses printing of the commands it generates and echoes a beep to the terminal after completion. Errors occurring during the process are still displayed. This was a lot easier to use concurrently during editing sessions than make itself.

X Window images

The X window illustrations in this report are the actual X images. They were obtained by first converting the windows to bit map files with the aid of a specially-written program:

```
bitimage [options] [filename]
```

The routine allows the user to select an area of the screen to be imaged by dragging a rectangle over the screen with the mouse. The bit map is then computed and written to standard output or a specified file. To demonstrate the effect of the action, the command shows the bit map by drawing it over the original image. The following options are recognized:

-w *ID* Image the drawable (i.e. Window or off-screen PixMap) with identification *ID* instead of interactively selecting a screen rectangle.

-a *left*x*top*x*width*x*height*
Image a specified area of the screen instead of interactively selecting a rectangle. With the **-w** option, image a specified area of that drawable. By default, the entire screen or drawable is imaged.

-p *plane* Use the specified bit plane of the image's pixel map as the bit map. By default, bit plane 0 is used.

-s Save original image—don't draw the output bit map over the original image.

The bit map files were then transferred to the Macintosh, where they were converted by another purpose-built program to a Clipboard picture:

```
Xbitmap [-r] [filename]
```

This MPW (Macintosh Programmer's Workshop) tool reads the data from standard input or a specified bit map file, and converts it into a QuickDraw picture which is put on the Clipboard. The picture may then be pasted into a word processor file, a ‘paint program’ for further editing, or whatever. The picture is scaled so that its size matches that of the image as it appeared on the screen it was captured on.

-r Don't scale the bit map but use the original resolution of the image. This will cause the image's size and aspect ratio to be distorted.

The Macintosh always retains the full detail of the image, so that each output device renders the image at its own highest possible resolution.

Interested parties may obtain the Xbitmap tool from the author.

2 Layout Manager

And where do we go?
Where do we go from here?

—With A Shout (Jerusalem), U2 [October]

The Layout Manager is the part of TRIP which handles requests from the Dialog and Editor Managers to build collections of Xr field editors or other undefined entities in an X window. It determines the optimal positioning of these editors with respect to one another and places them together in an Xr *panel*, which is then ready to be used by the Dialog Manager. Although entities other than Xr editors can also be positioned, they are not placed in the panel and must therefore be managed separately by the caller.

The Layout Manager consists internally of the actual layout-managing routines (in the layout.c, .h and .o files) and an additional independent set of region-handling routines (region.c, .h and .o). The region package might perhaps be useful to other applications as well.

Some of the Layout Manager routines are not actually used but have been provided to support future expansion. Notably, some of the advanced region-handling capabilities may be useful to programmers implementing new region evaluation functions.

LAYOUT MANAGER ENTITIES

The Layout Manager uses several entities in order to define and evaluate dialog layouts. The following types are defined for general use:

```
typedef enum {false, true} Boolean;     /* Boolean */

typedef struct {
     short      x, y;
     } Point;                           /* point */

typedef struct {
     short      left, top,
                right, bottom;
     } Rect;                            /* rectangle */
```

Note that this rectangle definition is different from that of Xr, which records its upper left corner and width and height.

Regions

Because regions are currently almost exclusively used by the Layout Manager internally, you may not need to read any of the region sections at all. The region capability is interesting, however, and available for general use.

A **region** is an arbitrary two-dimensional shape. Regions are used by the Layout Manager to assimilate and keep track of the screen area occupied by individual field editors in a dialog and for locating and evaluating unused areas of a layout for placing other field editors. The region concept in software was derived from QuickDraw [7, pp. I-141-142, I-181 ff.].

They are an extremely flexible way of representing shapes in a plane. They can have any form, size, consist of disjoint parts and even have holes in them. For example, both of the shapes shown below are regions. However, only the one on the right is likely to represent screen area occupied by field editors.

Figure 2-1. Regions

Regions are created, destroyed and changed by calling Layout Manager routines. For example, routines exist for creating empty regions (i.e., regions that contain no points), rectangular regions, and for creating the union, intersection, difference or exclusive-or of two regions.

Even though the definition of the region and the routines dealing with them are capable of supporting regions in the most general form described above, currently the only routines that create new regions 'from scratch' create either empty or rectangular ones. Regions are currently only used to represent dialog layouts, and the usefulness of, say, triangular or polygonal field editors has not yet been ascertained. Moreover, the X Window system supports only rectangular windows. The region-handling routines are especially efficient when dealing with regions composed of rectangles. However, should the need for different shapes arise, routines to create them can easily be written.

Although Release 11 of X Window (but not X10) also has a region definition capability and provides similar routines to deal with them, it not surprisingly has no built-in capability to allow an algorithm to evaluate the suitability of a particular region to represent a dialog layout. Unfortunately the data type in X is *opaque*, which means that the internal format of the region data is undefined from the X user's point of view and therefore cannot be accessed or analyzed by the Layout Manager.

Unless you are writing routines to create differently-shaped regions, or a layout evaluation routine (described below), there should normally be no need to directly access the region data structure.

The region representation and the routines operating on them unexpectedly proved to be extremely efficient and compact. In fact, some powerful routines which are not yet used by the Layout Manager have already been provided because their implementation was so straightforward. A region is represented by a variable of type `Region`, which is a pointer to a variable-length data structure:

```
typedef struct {
     short      size;           /* number of shorts */
     short      data[0];        /* region data */
     } *Region;
```

Size indicates the number of `short`s in the entire record, including size itself. The data describing the actual shape of the region is contained in the variable-length array data. Because the region data is an array of short variables, the maximum extent of the region is known to be at least the rectangle (–32768, –32768) - (32767, 32767)†, as guaranteed by the C standard. This should be sufficient to cover most display devices for some time to come. The region data format is shown with the aid of a small example:

† Depending on the hardware used. The system on which TRIP currently runs uses a 16-bit short

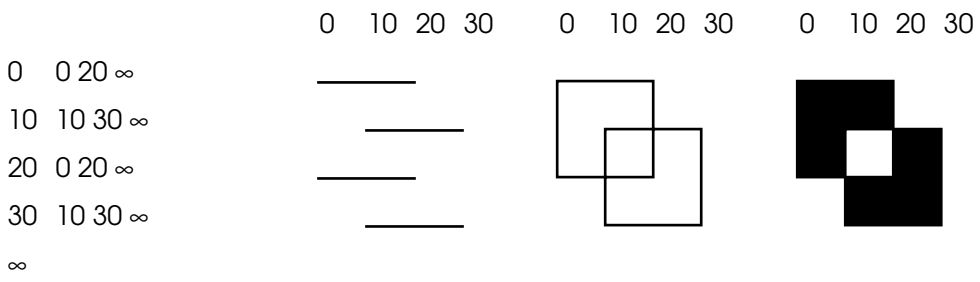

Figure 2-2. Region data

The region data consists of sequences of *y*-coordinates and **scan lines**. A scan line is a one-dimensional structure, itself consisting of a sequence of continuous and disjoint **spans**, or horizontal line segments:

```
typedef short ScanLine[maxScanLineSect + 1];
```

The ScanLine data type contains the beginning and ending *x*-coordinates of the consecutive spans in a scan line in increasing order. The definition of the upper bound of the array has significance only when variables of this type are actually allocated, as they are within some routines for temporary storage (RegionMunger and SectExtRgn, described below). Within the region definitions themselves the ScanLine type is only conceptual, since scan lines may actually have an unlimited number of spans.

> **Warning:** Because of the upper bound in the ScanLine array, some region operations *may* fail if a region changes state in more than 12 spans on a single scan line. This would not appear to be a problem as it is very unlikely that more than twelve disjoint editors could ever even be found on a single scan line. However, the upper limit can be adjusted by changing the value of the constant maxScanLineSect.

A scan line at a particular *y*-coordinate in a region specifies where the region changes 'state' on this scan line with respect to the former, i.e. from 'inside' to 'outside' the region or vice versa. Just as the representation itself is subdivided, nearly all of the region routines similarly reduce two-dimensional region problems to sequences of analogous one-dimensional scan line problems. The region representation is unique, so that two regions are equal if and only if the region data is equal and of the same length. The last span in a scan line and the last scan line in a region are delimited by a special value infinity (represented by '∞' in the figure), which is the largest representable short positive integer.

Although the region size field is not necessary for the region representation, it is used in region-handling routines to estimate the size of the result region when operations are performed on them.

The region representation derives its efficiency from the fact that it takes advantage of geometric locality of spatial objects [15, pp. 375, 561] by recording only the scan lines that change (*scan line coherence*) and then only the spans in the scan line which have changed (*edge coherence*).

Layouts

The Layout Manager defines and maintains data structures called **layouts**, which contain the information needed to build and run dialogs consisting of individual editors. Layouts, like regions, are created dynamically and referred to by pointers. They are created and maintained by calling Layout Manager routines so there should normally be no need to access them directly. Any number of layouts may be in existence at the same time, allowing more than one command editor to be active at once.

When a layout is first created it is **open**. Editors can be added to a layout only while it is open. After all the editors in the dialog have been added to the layout it must be **closed**. At this point, the Layout Manager completes whatever operations are necessary on the layout structure and makes the editors appear in the layout window. There is no way to reopen a layout after it has been closed.

Although the current implementation of the Command Manager makes no provisions for the grouping of parameters, one can see the advantages such an ordering would have. It would clearly be desirable for groups of related parameters to appear in a spatially coherent manner in the dialog. Without any parameter structuring facility, however, there is no guarantee that related parameters will appear anywhere near

one another. As the Command Manager imposes no structuring between parameters, it is also not possible for the Layout Manager itself to decide which parameters to take as related groups.

With a future extension of the Command Manager in mind, the Layout Manager already fully supports parameter structuring. Editors belonging to related parameters may be laid-out into a lower-level **sub-layout**. The sublayout representing the parameter group is then itself positioned in its entirety into the higher-level layout, ensuring that the related parameters stay correctly positioned with respect to one another.

As sublayouts may be opened within other sublayouts, there is no inherent limit to the depth sublayout nesting.

The Layout Manager builds command dialogs using a feature of the Xr library called *panels*, which are simply collections of field editors in a window.

A layout is represented internally by the following structure:

```
typedef struct {
      xrPanelInfo      info;           /* Xr panel information */
      xrPanel          *panel;         /* Xr panel */
      INT32            allocFields;    /* # of allocated panel fields */
      LevelLayout      *level;         /* current level in layout */
      } Layout;                        /* layout */
```

Info is an xrPanelInfo structure containing the definition for the panel[1]. It is assembled through specific Layout Manager calls which add editors to the layout. When the layout definition is complete the panel is created by passing this structure to Xr.

When the panel is created a pointer to it is returned by Xr and stored in the panel field by the Layout Manager. The field is NULL while the layout is open.

AllocFields contains the physical number of xrPanelField structures currently allocated for the layout. The xrPanelInfo structure contains, among other things, a pointer to a list of these structures—one for each editor in the panel. The number of xrPanelFields allocated is not necessarily the same as the number of editors. In order to reduce overhead the Layout Manager allocates a number of these fields at once so that most additions of editors do not lead to memory reallocation requests and therefore do not exacerbate fragmentation problems.

Level is a pointer to a linked list of LevelLayout structures. One of these structures is inserted at the head of the list when the layout is first created, and whenever a sublayout is opened. Thus, the LevelLayout pointed to by this field is always associated with the 'current' sublayout, and the last LevelLayout in the list corresponds to the main layout itself. When the layout definition for the current sublayout is complete, the information in the structure at the head of the list is propagated to the next higher level structure before it is removed. The structure is as follows:

```
typedef struct _LevelLayout {
      Region           rgn;            /* level's layout region */
      INT32            numFields;      /* # of editors in level */
      struct _LevelLayout *level;      /* next level in layout */
      } LevelLayout;                   /* layout */
```

Rgn is the region describing the union of the space currently occupied by all the individual field editors in this level of the layout. In other words, if a LevelLayout describes a sublayout, rgn will be fitted in its entirety into the next LevelLayout's region. When the sublayout is first created this field is initialized to an empty region.

NumFields is the number of field editors in this level of the layout.

Level is a pointer to the LevelLayout structure of the next higher level in the sublayout hierarchy. The structure at the end of the list (representing the entire layout itself) has level set to NULL.

Fitting editors

Besides the more general region-handling routines, the Layout Manager also uses a powerful routine for the optimal positioning of individual field editors within a dialog (see "Region fitting," below).

Basically, the routine works by scanning the region occupied by the editors which have already been placed for candidate locations for the new editor. It checks each of these locations to see if there is sufficient empty space available for the editor to fit. In most cases, it will even find 'hidden' space in holes inside the region. If the **extent** (the minimal rectangle completely surrounding a region) of the dialog region must grow in order to accumulate the additional editor, additional space for the dialog is 'allocated' in a specified **extension direction**.

The extension direction affects the final appearance of the dialog because it changes the way in which individual editors are located with respect to one another, as illustrated in the following figure:

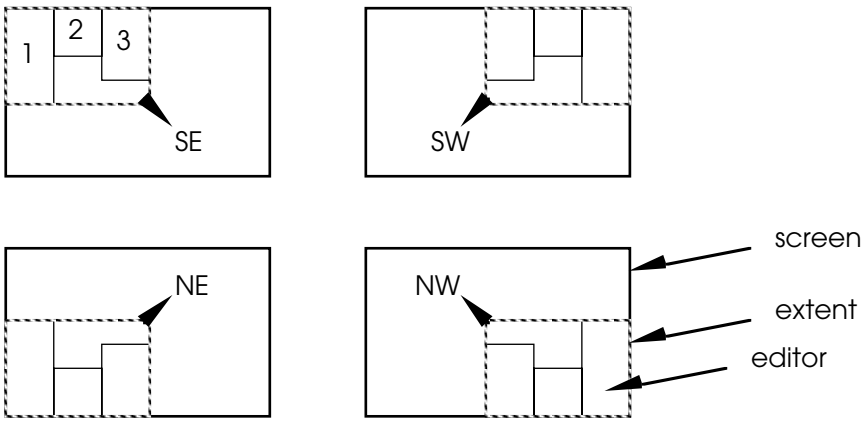

Figure 2-3. Extension direction

The most common extension direction would most likely be towards the southeast, which orders new fields as much as possible in the reading direction of Western languages, i.e., left to right and top to bottom. It might also be set according to a user's preference or to the writing direction of a particular language's script. For example, it would be more natural for readers of Hebrew to order the fields right to left and top to bottom, i.e., towards the southwest. Such a preference should be defined as an .Xdefaults variable [3] which can be defined separately by individual users of the program.

> **Note:** Because the Xr panel mechanism used by the Layout Manager requires that the upper left corner of the panel be located at the origin, the only currently effective extension direction is southeast. Other toolkits such as Xt, will most likely not have this asinine restriction.

Note also that the final appearance of the dialog is strongly dependent on the order in which editors are added to the dialog.

For each location which can accommodate the region occupied by the editor, the routine calls a **region evaluation function** supplied to it by the caller to select the 'best' of these locations. The final region and editor location are then returned to the caller. The region evaluation function 'looks' at a region representing a particular dialog layout and uses some criterion for determining its esthetic quality. It might, for example, take into consideration the size and aspect ratio of the region extent or the amount of empty space on the edges of and inside the region.

The region evaluation function currently used by the Layout Manager is a very small and fast one which rates regions on the basis of the area of the minimal *square* extent as well as the length of the perimeter of the actual extent. Using the area of the smallest square completely surrounding the region encourages tall regions to grow wider rather than taller, and wide regions to grow taller rather than wider. Incorporating the perimeter of the actual extent gives preference to smaller (more compact) regions.

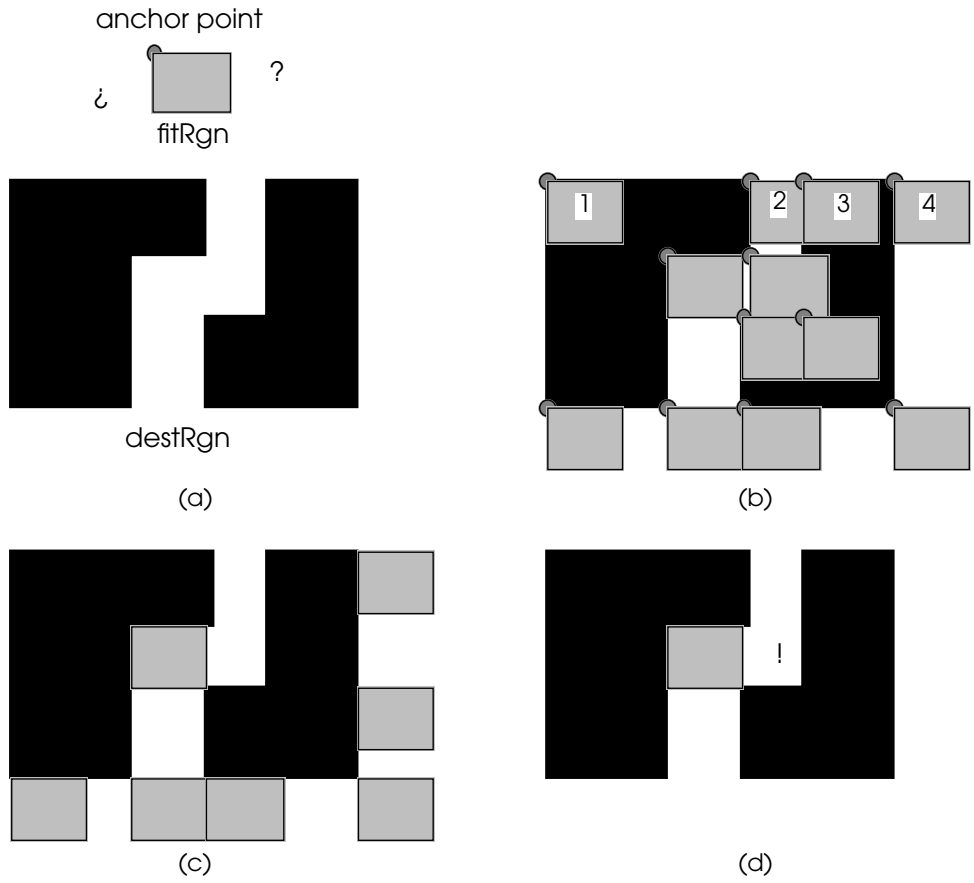

Figure 2-4. Region fitting selection process

Fitting regions

The Layout Manager uses a simple but effective algorithm for finding space for a new region (called the *fitRgn*) in an existing region (called the *destRgn*). Refer to figure 2-4(a).

Depending on the specified extension direction, the fitRgn is assigned an **anchor point**. This point is the corner of the fitRgn opposite to the extension direction. In the figure the extension direction is southeast, so the anchor point is the northwest corner of the fitRgn. The anchor point is matched with each internal corner of the destRgn in turn. (b.)

In such a location, if the fitRgn overlaps the destRgn in any area the fitRgn is shifted horizontally in the direction nearest to the extension direction (i.e., east or west). This enables it to be placed along the destRgn even if it originally overlapped. The fitRgn is shifted until it fits, or until any previously nonoverlapping parts of the fitRgn start to overlap the destRgn (c.) This prevents the fitRgn from floating over the destRgn into an area where fitRgn placement location candidates have already been selected. For example, in 2-4(b), the regions marked 1, 2 and 3 would each end up with region 4. Note that in rare cases the algorithm will not correctly find empty space because it does not attempt also to shift the region into its vertical component extension direction. I cannot see a way of doing this without significantly slowing down the algorithm, and the potential benefits seem marginal.

Finally, all resultant locations are evaluated and the region with the highest evaluation score is selected as the appropriate location. (d.) Confirm that the operations described here ensure the destRgn is extended only in the assigned extension direction.

Although this simple method is adequate for regions without many edges, its efficiency is bound to be too low when operating on regions with, for example, curved sides.

ROUTINES

Besides regular testing on the HP 9000, the region routines also ran successfully without modification on an Apple Macintosh.

Region routines

The region package does not need to be initialized. Routines that create new regions return NULL for the region if the space for it could not be allocated. It is not legal to pass NULL values for region parameters.

```
Region NewRgn();
```

Returns a new empty region, i.e., one that contains no points.

```
Region NewRectRgn(int left, int top, int right, int bottom);
```

Returns a new region containing only the points within the rectangle defined by the given four boundary coordinates. If the specified rectangle is an empty rectangle, i.e. one with right ≤ left or bottom ≤ top, it creates an empty region.

```
void DisposeRgn(Region r);
```

Disposes of the memory associated with the region. Call this routine after you're completely done with the region. After calling DisposeRgn, the region may not be referenced again.

```
Region CopyRgn(Region r);
```

Creates and returns a new copy of the region. The new region may be used independently and without affecting the original.

```
void CornerRgn(Point *corner, Region rgn, short direction);
```

This routine examines the given region and stores the location of one of its corners in corner. The particular corner returned depends on direction, which should be one of the four direction constants defined for the FitRgn routine, below. If the region is empty, CornerRgn always returns the origin (0, 0).

Since regions need not be rectangular, the definition of a corner is somewhat ambiguous. The rule is that CornerRgn first looks as far north or south in the region as possible, and then extends as far east or west as possible. The result is that CornerRgn may not return the eastern- or westernmost point in the region. To find the eastern- or westernmost coordinate in a region, use ExtentRgn.

```
void ExtentRgn(Rect *extent, Region rgn);
```

Returns in extent the smallest rectangle completely enclosing all the points in the given region. If the region is empty, ExtentRgn returns the empty rectangle (0, 0)-(0, 0).

```
void OffsetRgn(Region rgn, Point offset);
```

Changes the specified region so that all the points in it will have been translated by an amount offset with respect to the original region.

Note that OffsetRgn changes the region data of the given region. To keep the original region, call CopyRgn first and then pass the copied region to OffsetRgn.

```
Boolean SectRgn(Region rgnA, Region rgnB);
```

Returns TRUE if the regions intersect at any point, i.e., if any point in the coordinate plane is a member of both regions.

SectRgn calls the more general internal routine SectExtRgn.

```
Region UnionRgn(Region rgnA, Region rgnB);

Region SectionRgn(Region rgnA, Region rgnB);

Region DiffRgn(Region rgnA, Region rgnB);

Region XorRgn(Region rgnA, Region rgnB);
```

Each of the above routines create and return a new region containing either the union, intersection, difference or exclusive-or of the two given operand regions, as illustrated in the following figure. RgnA and rgnB may be the same regions.

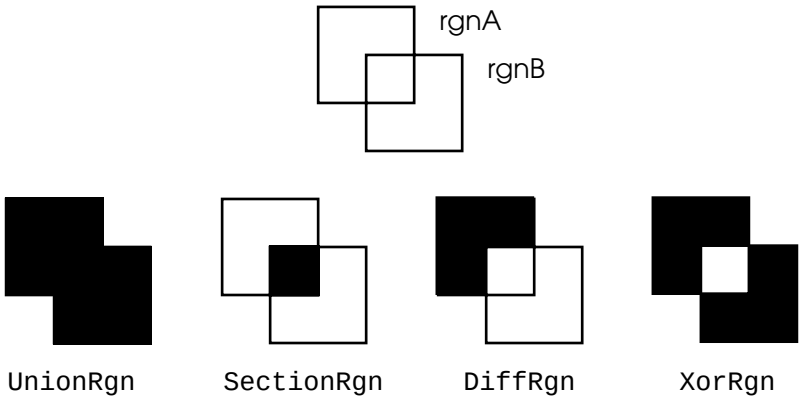

Figure 2-5. Operations on regions

The routines call a common internal routine RegionMunger with an additional opcode specifying the particular operation to be performed. RegionMunger temporarily allocates space twice the size of the operand regions (the maximum possible size of the output region) but then reallocates the space to match the actual result region size.

```
Region FitRgn(Region destRgn, Region fitRgn, Point *offset, short
         extend, int (*eval)());
```

Calculates the optimal union region, according to a specified region evaluation function, of a given destination region and an additional region to be 'fitted' into it so that it nowhere intersects the destination region. It returns a new region representing the union of the two regions. FitRgn is called by EditorLayout and RgnLayout when adding new field editors to the layout, to find the best place for it in the dialog.

DestRgn specifies the destination region—this region will typically represent the space occupied by the field editors which have already been placed in the dialog. FitRgn specifies the space required by the additional editor. It is not necessary for either destRgn or fitRgn to ‘originate’ at (0, 0). If the destination region is empty, one of the corners of the fitRgn will be located at the origin in the union (which one depending on the value of extend).

Upon return, the offset will indicate the amount of offset which has been applied to fitRgn to move it to its location in destRgn. For example, if a certain point of fitRgn is located at (0, 0) the same point of the fitted region in the union can be found at offset.

Extend specifies the direction in which the destination region may be expanded. One of the following constants should be used:

```
#define northWest 0         /* towards northwest */
#define northEast 1
#define southWest 2
#define southEast 3
```

Eval is a pointer to a region evaluation function of the form

```
int MyRegionEvaluator(Region r);
```

Given a region representing a dialog layout the region evaluator should return an integer value indicating its desirability, with higher numbers indicating a more desirable layout. The numbers returned by the function may be negative, but should exclude the single most negative integer.

Layout routines

Before using any of the other layout routines the Layout Manager should be initialized by calling InitLayout. This sets some global variables used internally.

New open layouts may be created by calling NewOpenLayout. Editors can be added while it is open by calling EditorLayout. The layout may subsequently be closed by calling CloseLayout. When the dialog is no longer needed, the layout may be disposed of together with its editors by calling DisposeLayout.

Sublayouts may be started on open layouts or sublayouts by calling OpenSubLayout. Any editors added using EditorLayout between this call and CloseSubLayout are added to the sublayout.

RgnLayout may be used instead of EditorLayout when space for a region needs to be reserved in the layout without adding an editor. This is useful, for example, when the region is not occupied by an Xr field editor, or if the editor does not run in the same process and therefore cannot be added to the panel.

```
void InitLayout();
```

Initializes some internal variables of the Layout Manager.

```
Layout *NewOpenLayout(Window window);
```

Creates and returns a new open layout in the specified window, or NULL if it could not be created.

For best results, the window should not be mapped until after the layout is closed, because the window will be moved and resized at that time. To avoid disconcerting the user, the system should not appear to be changing the location of windows on the screen of its own accord.

```
xrPanel *CloseLayout(Layout *layout);
```

Closes the specified layout. An Xr panel is created containing the editors in the layout, which are then displayed in the layout window. Regardless of its previous size, the window is resized to completely surround the editors and is centered on the current display.

If the operation was successful, a pointer to the newly created panel is returned by the routine. This pointer is needed to gather input from the panel or otherwise control the editors in it. Note that once the layout is closed the Layout Manager does not touch the layout or the editors themselves at all until the layout is disposed of.

Layouts to which no editors at all have been added do not cause a panel to be created. In this case, or if any error occurred, NULL is returned by the routine.

It is not possible add editors or regions to a layout after it has been closed.

Excess allocated xrPanelFields are also reclaimed at this time.

```
void DisposeLayout(Layout *layout);
```

Disposes of a layout and all of the structures associated with it, and removes the panel with its editors from the layout window. The layout window is left unchanged. Call this routine after the layout is completely done with.

DisposeLayout can also be used with an open layout to release any resources occupied by it, if, for example for some reason construction of the layout could not be completed.

```
Boolean OpenSubLayout(Layout *layout);
```

Opens an empty sublayout in the specified open layout, returning TRUE if successful. Any editors or regions added to the layout while the sublayout is open will be laid-out as a group into the next-higher level (sub)layout.

Every call to OpenSubLayout must be balanced by a call to CloseSubLayout, or CloseLayout will fail.

```
void CloseSubLayout(Layout *layout);
```

Closes the last opened sublayout. The editors in the sublayout are 'moved' to and fitted into the next higher-level (sub)layout.

It is absolutely allowable to call OpenSubLayout again after calling CloseSubLayout. This will create a different new empty sublayout at the same lower level.

```
Boolean EditorLayout(Layout *layout, xrEditor *(*editor)(), void
          *editorInfo);
```

Adds an Xr field editor to the open (sub)layout and returns TRUE if it was successful. Editors are not displayed until the layout is closed.

Editor is a pointer to the Xr editor function such as XrScrollBar. All the Xr editors have a common entry call of the form

```
xrEditor *XrEditor(xrEditor *instance, INT32 message, INT8 *data);
```

It is possible to write your own field editors which may be added to a layout, by carefully following the Xr protocol for field editors [1, appendix B] and passing a pointer to the entry point of your editor to EditorLayout.

EditorInfo should point to an appropriate editor information structure (e.g., xrScrollBarInfo for an Xr scroll bar), which must be completely initialized except for the editor rectangle origin and the editor window. The size of the editor rectangle should already have been determined for the editor, either by calling the editor routine with MSG_SIZE to let the editor calculate its own size or by setting it directly. (In the latter case it is probably judicious to send the editor MSG_SIZE anyway to ensure the specified size can be supported by the editor.) The structure contents will be changed by EditorLayout so as to correctly position the editor within the dialog.

It is important that the editor information structures remain in existence until the layout is disposed of. The Layout Manager will not automatically release any resources used by the information structures. Also, any such structure may be used in only one layout at a time.

```
Boolean RgnLayout(Layout *layout, Point *offset, Region region);
```

Adds the region to the specified layout without adding an editor to the layout panel. Returns TRUE if successful. Offset is set to the amount by which the region was translated in order to fit in the layout.

This routine is useful when the specified region is not or cannot be associated with an Xr field editor, or in general to avoid certain regions of the layout being occupied by editors.

Debugging routines

The following two routines are provided for debugging purposes, and are compiled only if the symbol DEBUG is defined at compilation. This may be done directly in the source file by writing

```
#define DEBUG
```

or from the UNIX command line by specifying the -DDEBUG option to the cc command.

```
void PrintRgn(Region r)
```

Prints all the scan lines with their *y*-coordinates in the region, that is, each scan line where the region changes.

```
static void *PrintScanLine(ScanLine *theScanLine)
```

Prints the spans in the scan line and returns a pointer to the first byte following the last of the scan line data. If the scan line was part of a region, this would be the *y*-coordinate of the next scan line in the region.

Internal routines

Next, a few of the more important routines used internally are highlighted.

```
static void *DeltaScanLine(ScanLine new, ScanLine old, ScanLine
        delta);
```

Returns the exclusive-or of scan lines old and delta in the scan line new. In other words, new is the scan line which results when the spans in delta indicate where old changes state. Old and delta should be valid scan lines, that is, their spans must be minimal (i.e., have no two distinct adjoining spans) and sorted in order of increasing *x*-coordinate. DeltaScanLine correctly handles cases where spans in old and delta combine to form larger single spans or cancel each other out.

The routine returns a pointer to the first byte following the end of new. This is especially useful when the routine is used to create new region data, in which a new scan line will immediately follow the scan line just returned.

DeltaScanLine is used by various routines to keep track of the current ‘state’ of a region at a particular *y*-coordinate while scanning through the region's region data.

> **Warning:** The routine does not check to see if all the spans of the result scan line actually fit into the array. At worst it may require less than twice the storage required by the two input scan lines.

```
static void *ScanLineMunger(ScanLine slny, ScanLine sldy, ScanLine
        sloy, ScanLine sla, ScanLine slb, RegionOperation op);
```

Computes the new current and changes scan lines which result if the specified region operation is applied to two region scan lines.

The new current result region scan line is returned in slny. The state changes of the result scan line with respect to its previous state, which must be passed in sloy, are returned in sldy.

Sla and slb should specify the current states of the region A and B scan lines. Op specifies the operation to be performed on the scan lines, but the routine itself processes scan lines in precisely the same manner regardless of its value. The RegionOperation data type is defined internally as follows:

```
typedef enum {rgnUnion, rgnSection, rgnDiff, rgnXor} RegionOperation;
```

The routine returns a pointer to the first byte following the newly returned data in sldy, which, again, is useful when it is being used to create new region data.

```
static Region RegionMunger(Region rgna, Region rgnb, RegionOperation
        op);
```

Creates and returns a new region by applying the specified operation on the operand regions, or NULL if the region could not be allocated.

RegionMunger temporarily uses twice the storage occupied by the operand regions.

```
static Boolean SectExtScanLine(ScanLine ssla, ScanLine sslb, int
        *shift, int offset, short extend);
```

Returns TRUE if the two given scan lines intersect at any point, or FALSE if they don't. Offset specifies the amount by which scan line B should be considered to be offset, a positive number indicating a shift to the right.

If the scan lines intersect and extend is not equal to nowhere, SectExtScanLine attempts to make the scan lines nonintersecting by shifting scan line B so, that any of its spans that intersect A are moved out of it. It will not, however, shift B if any previously nonintersecting spans of B would become intersecting *during* the shift. The direction in which scan line B may be shifted (east or west) should be specified in extend, using one of the four extension directions or the additional constant

```
#define nowhere -1
```

if no shifting is to be attempted. The minimum additional amount, above that already specified by offset, that scan line B needs to be shifted in order to make it nonintersecting is returned in shift, where a positive value indicates a shift to the right. Zero is returned if the scan lines cannot be made nonintersecting without moving nonintersecting spans of B across A.

```
static Boolean SectExtRgn(Region rgna, Region rgnb, int *shift, Point
        offset, short extend);
```

Returns TRUE if the two given regions intersect at any point, or FALSE if they don't. Offset specifies the amount by which region B should be considered to be translated.

If the regions intersect and extend is not equal to nowhere, SectExtRgn attempts to make the regions nonintersecting by translating region B horizontally, so that any parts intersecting region A are moved out of it. It will not translate region B if any previously nonintersecting parts of B would become intersecting *during* the translation. The direction in which region B may be shifted (east or west) should be specified in extend, using one of the four extension directions or ‘nowhere’ if no shifting is to be attempted. The minimum additional amount, above that already specified by offset, that region B needs to be translated in order to make it nonintersecting is returned in shift, or zero if the regions cannot be made nonintersecting without moving nonintersecting parts of B across A.

FUTURE IMPROVEMENTS

Future versions of the Layout Manager might attempt to expand the regions occupied by individual editors to fill up any empty space in the layout region. It is not certain that this will actually be beneficial to their appearance.

The predefined region evaluation function is a very simple. Putting some thought into another version may offer the possibility of greatly improving the appearance of layouts in dialogs.

3 Editor Manager

So far
Should I take it just a little farther?
So good
And if I think about it, it gets better

I Could Get Used to This, 'Til Tuesday [Voices Carry]

The term 'Editor Manager' as used in this document describes a separate module used by the Dialog Manager. This is a departure from the use in [8] where it encompasses the entire dialog management system. The Editor Manager translates requests from the Dialog Manager for editors of certain data types into the creation of appropriate Xr field editors (called *internal* editors) or user-defined editors contained in separate compiled units (*external* editors).

Editors are routines which display graphical entities on the screen representing a particular data type value, and allow a user to interactively view and modify this value using a mouse and keyboard. Often there will be more than one **instance** of the same editor type in existence at any one time. A specially-designed window containing a number of editor instances to allow the specification of all the parameter values of a particular simulator command is called a *dialog.*

The Editor Manager takes care of starting, stopping and generally managing additional processes which may be needed to run editors. This naturally includes taking care of setting up and running the communication links between these processes and the Editor Manager.

USING THE EDITOR MANAGER

Particular types of editors are created upon request. Once an editor is created, it is assigned a positive integer **editor reference number** which may be used by the caller to control it. There is no theoretical limit to the number of editors which may be in existence at any time.

At this point an initial value for the editor may be specified, or a parameter-specific default will be used. Editor values are obtained as the result of earlier invocations of editors—the caller never creates editor values itself. Their format is defined by the type of data the editor handles so that they can later be used to generate the output which is used to feed the simulator.

The editor then stays in existence and handles input from the user as appropriate. The caller must specifically ask for the editor's value when needed. It is undetermined whether a user has altered the value of the editor in the meantime. If the editor is one of several in a window, it is the caller's responsibility to query each of the editors when necessary.

When the caller is done using the editor it must explicitly remove it. The previous usage of the editor reference number is thereby rendered invalid, but it may be assigned to another new editor instance in the future.

Editor classification

Editors come in many different shapes and sizes. First of all they can be distinguished on the basis of whether they are **command** or **parameter** editors. Commands typically consist of several parameters, and likewise, anything a single command editor can do could also be done (albeit probably less effectively) by a number of parameter editors. A command editor may in some cases be more useful or intuitive than a respective set of parameter editors.

Command and parameter editors can be classified further into editor **variants** according to the kind of data they edit. For a command editor this will be a particular command (e.g., “Add-Material-Region”) and for a parameter editor some data type (such as “integer”, or “temp” for temperature).

Thirdly, a distinction can be made between **internal** and **external** editors. Internal editors are those contained in the Editor Manager itself, and have been chosen so that the great majority, if not all of the command and parameter types can be reduced to the types of data handled by these editors. External editors may be written by other programmers to facilitate data entry for certain commands or data types of a particular simulator system. They are contained in separately compiled files which may be modified, or added or removed from the system at any time.

Editor programs can support many different editor variant types. The ‘internal editor program,’ the Editor Manager itself, has five internal parameter editors:

- An **int** variant parameter editor, which allows users to specify any integer value;
- A **real** variant parameter editor, for the specification of floating-point numbers;
- A **string** variant editor, for the specification of character strings;
- A **boolean** variant (note no capitalization) editor, which allows the specification of one of two states;
- A *list* variant editor, which allows the user to make a single selection from a predefined list of items. There are actually a great number of different list variants depending on the items in the list, which are all handled by the same editor.

Contrary to initial intentions the internal editors (or *default editors*, as they were also called) do *not* run in a separate process. Although this would indeed simplify the implementation, incorporating them into the existing TRIP program saves one additional long-term process and the concomitant interprocess communication.

The appearance of these editors in a window is illustrated in the following figure:

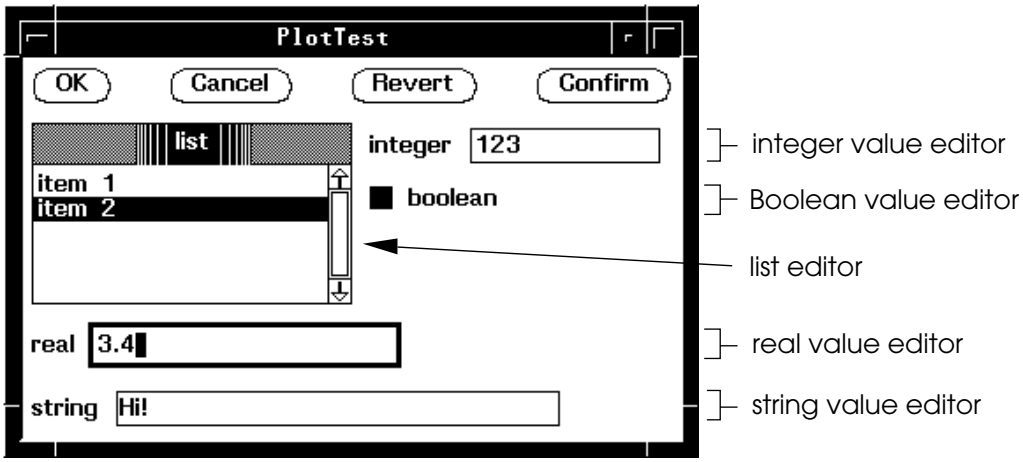

Figure 3-1. Internal editors

The internal parameter editor variants correspond to the five *primitive types* defined in [8, p.19]. Internal command editors are not strictly necessary because commands should always be reducible to their constituent parameters. Note that this is not the task of the Editor Manager—it operates only on the level of individual editors and any necessary decomposition of commands into parameters must be performed externally by the Dialog Manager. In fact, the Editor Manager treats command and parameter editors absolutely identically, although the editors themselves obviously handle different kinds of data.

Editor specification

When creating new editor instances, the Editor Manager passes the editor a *command* or *parameter specification* derived from the text in the user-defined CDD file. It is up to the individual editors to gather any additional required information from the editor specification. Parameter editors receive parameter specifications of the form

```
<par> ::= '<' <label> : <type> : <value> '>'
```

(see the remark on BNF notation in the Command Manager chapter).

A parameter **label** is a non-editable string which should appear somewhere in the editor instance to help the operator identify individual editors with particular parameters., as well as differentiate between multiple instances of the same editor variant type in a single dialog. For instance, there may be four integer variant editors in a dialog to specify the coordinates of a rectangle, differing only in their labels.

Type is the type of the data to be edited. The syntax of this string is defined by the editor that manages the type. The internal parameter editor types are the variants listed above, ‘int’, ‘real’, ‘string’, ‘boolean’ or a list type. List types are many and are specified by means of a single-quoted list of items, separated by commas:

```
<list type> ::= ' <item> '{' , <item> '}' '
```

The brackets indicate the list may have any number of items in it.

Lastly, the value specifies a default editor value, which is a string containing a readable representation of the value of the parameter. When an editor is created it may be passed an initial value, but if it isn't the editor should use the value specified here. This allows the same editor to have defaults depending on the command parameter, not just on the variant. When requested to do so, the instance will return a similar string containing its current value.

For a list variant editor, a legal value is any one of the list items single-quoted. For a boolean variant editor it is either “true” or “false”. Int, real, and string variant editors accept the ASCII representations of any integer, real or string value, respectively. If a specified value is invalid, the actually used default value is editor-specific.

Command editors receive, not unexpectedly, command specifications. They are basically concatenations of parameter specifications with some additional parameter repetition constructs. Refer to [8, ch.4] and the Command Manager chapter for more detailed information about command specifications.

To exemplify, the editor specifications of the internal editors in figure 3-1 were:

```
< list : ' item 1, item 2' : 'item 2' >
< real : real : 3.4 >
< integer : int : 123 >
< boolean : boolean : true >
< string : string : Hi! >
```

If there were a command editor for this command, it would receive all five parameter specifications as the command specification.

> **Note:** Sometimes the specification passed to an editor may not be what it expects. For instance, if the CDD file specifies a parameter with a type of “temp” defined as a subtype of integer, it will end up with the int variant editor if the temperature editor is not available. Programmers writing external editors should keep this in mind, as their editors may find themselves in a similar situation. List variant editors are currently safe from this happening, as the Command Manager does not allow subtypes to be defined from list types.

EXTERNAL EDITORS

External parameter editors replace internal ones to provide a more natural way of editing certain types of data. External command editors do the same, but replacing all of the parameter editors for a particular

command by a single integrated editor. For example, it would probably be easier to draw a polygon than to enter the individual vertex *x*- and *y*-coordinates.

Because external editors are contained in separate source files they can only run as separate processes. When the Editor Manager first receives a request for an external editor, it starts a new process to run the program handling the editor. The name of this program is specified by the user in the CDD file (see also the Command Manager chapter). Multiple processes handling different types of editors may be running at the same time. Requests for additional editors which are handled by a running program do not of course cause the creation of more processes.

Communicating with the Editor Manager

Originally it was thought that communication between the Editor Manager and editor processes would need to occur through UNIX pipes. Pipes are designed primarily to transfer large (or at least substantial) streams of continuous data. However, communication between the Editor Manager and editor processes consists mainly of short bursts of very small exchanges (usually less than 25 bytes) of data. Using pipes in this instance would appear to be a waste of system resources considering the long periods of time that they remain in idle existence.

Fortunately, the X Window package provides a more appropriate solution in the use of **properties**, which are variable-length packages of data associated by an X client to a particular window on the system. Properties are used for instance, by X applications (such as TRIP) to make certain configuration preferences available to a window manager. When they are created, a property is referred to by its **name**. Afterwards, they may be read or changed by any client that knows the property's **atom**, which is an integer property identification. By communicating appropriate atoms to editor processes at suitable times the Editor Manager and editor processes have an extremely convenient method of communication available to them. Note that there is no concept of a connection or channel being in existence for extended durations, other than that between the application and the X server, which is always open anyway.

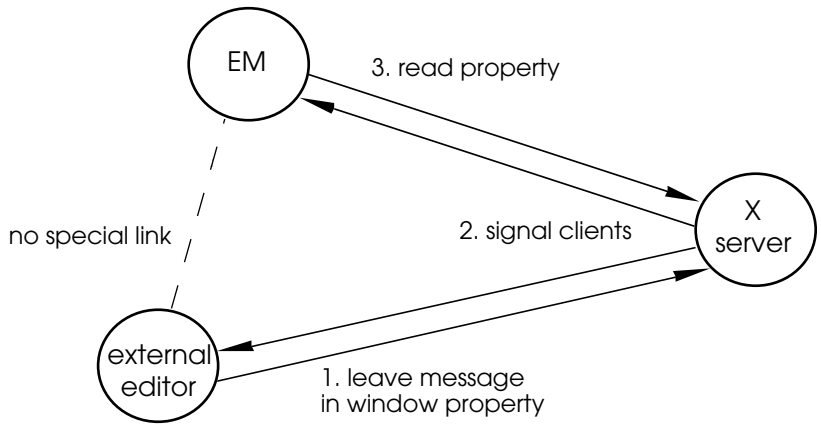

Figure 3-2. External editor communication using properties

Communication in the opposite direction proceeds likewise. Whenever a particular property is set or changed the X server sends a PropertyNotify event to any client having set itself up to receive them, allowing the Editor Manager/external editor communication to be incorporated into the normal stream of event processing.

This actually eliminates another problem which would occur if pipes were used, due to the fact that there would then be two independent input-gathering mechanisms (pipe reads and X events). If no input is available on one, the other one must be checked, and so forth—special care would have to be taken to ensure that a *busy-waiting* situation was not created in which valuable processor time is wasted just looking for input. With window properties we still get all our input from a single source so that a process waiting for input can simply be blocked when none is available, allowing another process to run instead.

Protocol

Communication through window properties occurs on two levels. First of all, whenever an editor process is started, a **process property** is associated with it. The property's name, which will be of the form "TRIPexp*nnn*" is passed to the process in its main() entry procedure argument list. Second, an **instance property** is associated with each external editor instance. These property names are likewise of the form "TRIPexe*nnn*", where *nnn* is the editor reference number. The editor process receives instance property names through the process property. All these properties have been arbitrarily associated with the root window, because editor processes may have editor instances in any number or no windows. Because atoms are assigned by the X server on the basis of the property names and are thus guaranteed to be unique, there is negligible risk of 'property collision.' Note that because editor reference numbers are reused, instance properties may have applied to different editors during their lifetimes. The same holds for process properties created by the Editor Manager. Recycling properties is actually a good idea, as they cannot ever actually be completely removed from the system once they are created. (Even though the following discussion and X terminology say that properties are deleted, they are not really.)

An description of the six different phases in the protocol is given next.

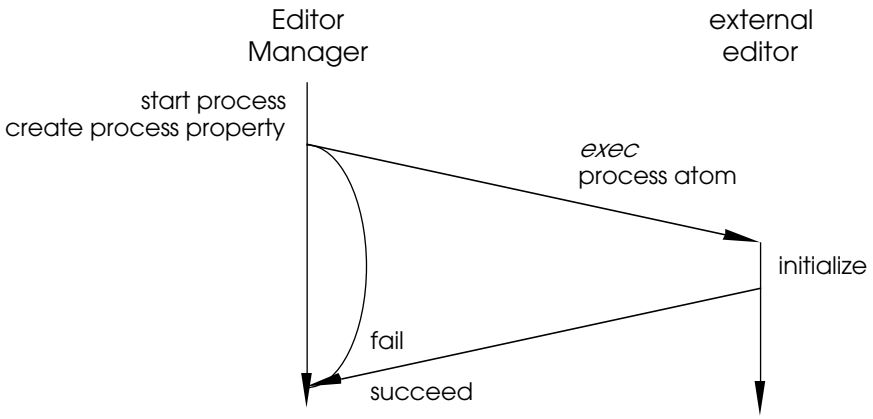

Figure 3-3. External editor process creation

- *Process creation.* The editor process is forked and exec'ed. The process property name is passed to the process as the first argument argv[1]. The process will signal the Editor Manager through the process property as to whether the process creation and program initialization were successful. If the exec failed and the editor program was therefore never reached, the Editor Manager notifies itself using the same mechanism.

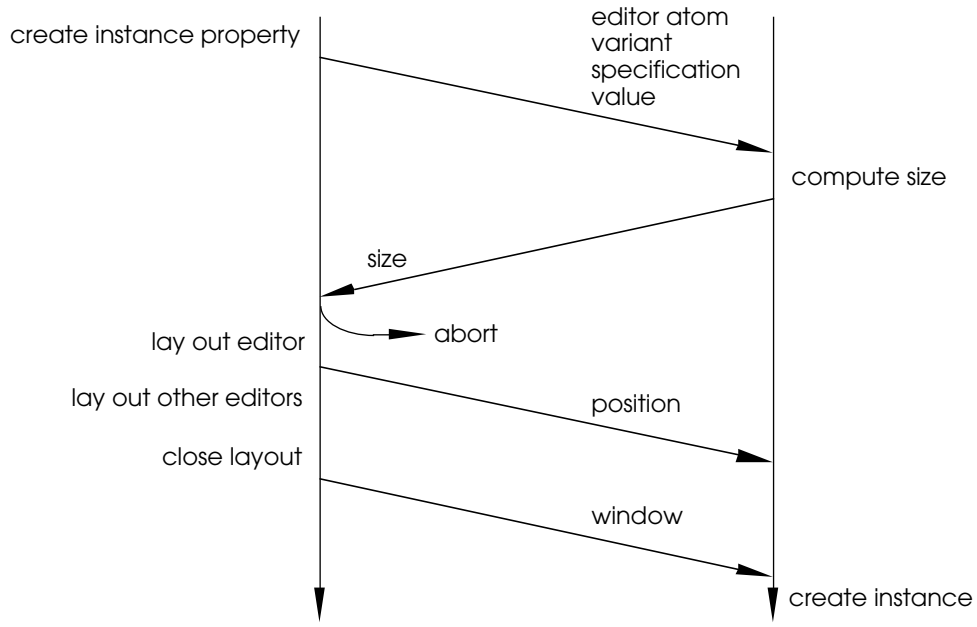

Figure 3-4. External editor instance creation

- *Instance creation.* The Editor Manager places a data package consisting of the new editor instance's property name, the editor variant, specification and initial value, into the process property. The editor process is thereby automatically notified that a new instance is requested and reads the information out of the property. All further communications concerning the new instance proceed through the instance property. If it decides to create the instance, the editor process passes the size of the editor back. Otherwise, it indicates that the process will not create the instance. The process is then never at all contacted through the new instance property concerning this instance.

  Otherwise, the Editor Manager places the editor in the open dialog layout and sends the new editor's position to the process. Finally, after the layout is closed all external editor instances are notified of the ID of the newly created panel window in which they should create windows to display themselves. Editors cannot use the panel window itself because it is not possible for two X clients to receive mouse button events on the same window.

  For the duration of the instance's existence, excepting the following three explicit phases in the protocol, the editor instance operates completely independently from the Editor Manager. It receives and processes its own input events and keeps track of the current state of the instance.

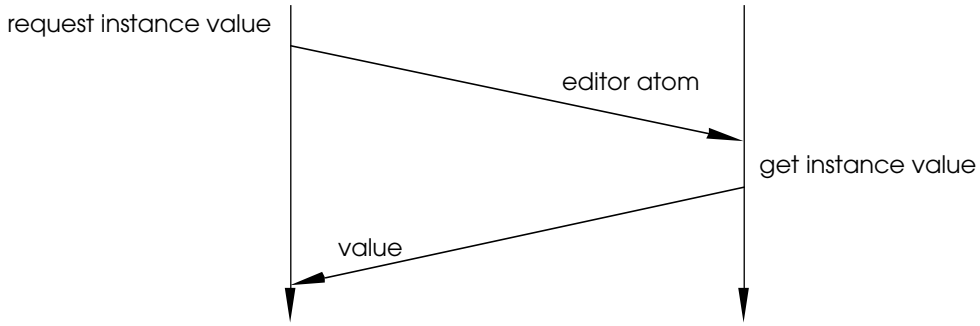

Figure 3-5. Instance value query

- *Value query.* Whenever necessary, the Editor Manager requests an instance to report its current value. This might happen in response to a request from the operator or an indirect explicit request from the Dialog Manager. The instance never reports its value of its own accord.

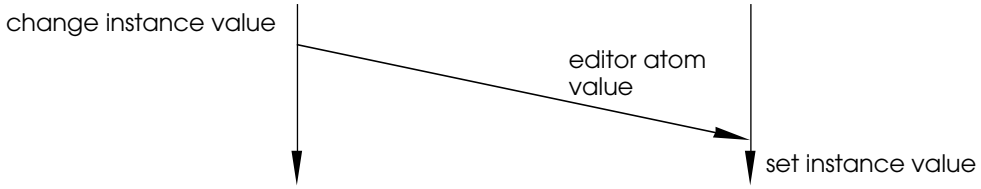

Figure 3-6. Instance value modification

- *Value modification.* Likewise, the instance may be requested to change it current value. It is automatically reported through a special event type.

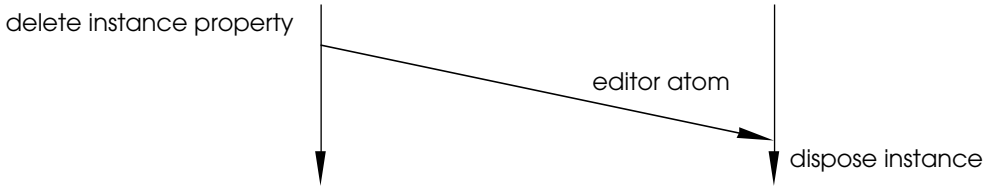

Figure 3-7. Instance disposal

- *Instance disposal.* When the instance is no longer needed the Editor Manager simply deletes its instance property. The editor process receives an event notifying it accordingly and removes the instance.

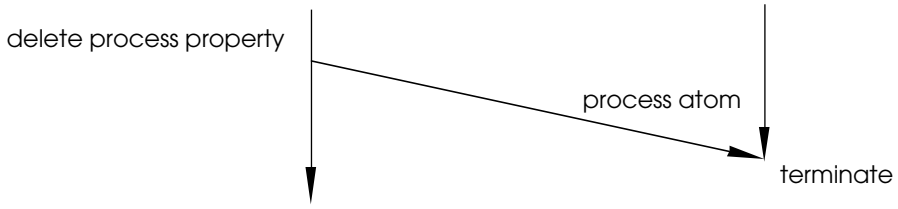

Figure 3-8. Process disposal

- *Process disposal.* Likewise, when the process is no longer needed its process property is deleted. Currently, this happens when all its instances have been disposed. In future, the Editor Manager might not immediately terminate the process but decide to keep it in a latent state for a while in anticipation of new instances being created in the near future.

Writing external editors

Those wishing to write external editors need not bother with the details of the protocol described above, because a small interface module has been provided to manage the low-level protocol. This module, ‘extedit’, should be included and linked with all external editor programs. Should there be any future changes to the protocol, the editor should then remain at least source-code compatible.

A sample external editor ‘temp’ has been provided. This program contains a ‘temp’ variant temperature parameter editor. This very crude example editor was used for validation of the protocol. The fact that it supports only one instance at a time was also useful in testing of abnormal protocol termination. The source can be used as a guide to external editor writing.

External editor programs can be written to handle more than one different variant of editor, or can combine both command and parameter editors. There is certainly a very good reason to want to consolidate as many different editor variants into a single program as possible, as the overhead of the graphics libraries used may be considerable. For example, using both the X and Xr libraries adds about 250 kilobytes of object code to the editor program. Note however, that there is no constraint on which toolkit is used by an editor program. In fact, the example editor does not use any toolkit at all.

Once an editor program has been written, the CDD specification file must be changed to make TRIP aware of its existence. As of yet, however, the Command Manager does not support the specification of

either parameter editors or sublevel command (option) editors. [8] Describes the current CDD file syntax, and how to specify the use of an external editor program.

Although the protocol provides a mechanism to deny requests to create editor instances, editor processes really should rarely need to use this feature. The Editor Manager and operating system have expended considerable effort to reach this point in the protocol, and processes routinely denying requests could have a substantial detrimental effect on the responsiveness of the system. The operator will not appreciate having to wait for extended periods of time (especially when the system and network are congested) and may intuitively learn or purposely choose to avoid using certain features.

Specific editor instances may be uniquely identified by means of their instance property atoms. However, the extedit interface also allows individual instnaces to be discriminated through a **reference constant** which may be assigned to them by the editor program when they are created, and read or changed whenever desired. Such a constant would typically be used to store a pointer to a private data structure, but since it is not accessed by extedit it may be used in whatever manner the editor program desires. Extedit internally maintains a variable-length table of instance atoms and reference constants.

Programmers writing external editors should consider the use of colors in their editors. For instance, the sample external editor 'temp' draws the setting in red. Although color can greatly enhance the efficacy of an editor and make its appearance more pleasant, color should never be the only way that items in an editor can be distinguished, because the program should also be usable in a black-and-white environment.

Editor Manager events

The protocol interface module relies on being passed events which have been received by the editor program. It is absolutely essential that anywhere in the program where events are removed from the event queue they are first passed to extedit before being processed. This gives the protocol handler a chance to examine the event and determine whether it applies to Editor Manager communication and take any additional action. Currently only PropertyNotify events need to be intercepted.

If the event does apply to Editor Manager communication, the event type and record contents will be modified so that the editor program is more accurately informed of the nature of the event. For this purpose a number of new Editor Manager event types have been defined. This method has the distinct advantage of unobtrusively mixing in with the normal process of event gathering—an external editor program reacts to the new event types in just the same way as to the existing X events.

The new event types and their use are explained in turn.

EError

```
typedef struct {
     XPropertyEvent   xproperty;       /* event that caused the error */
     } EErrorEvent;
```

Returned whenever an error occurs in the communication protocol. As this event should actually never occur, the correct action to take in any particular circumstance is undefined. The safest course to follow would probably be to continue to respond only to instance and process disposal events. The event type is currently identical in contents to an X PropertyNotify event.

EQuit

```
typedef struct {
     XPropertyEvent   xproperty;       /* event that caused this action */
     } EQuitEvent;
```

Signal to the external editor program to terminate. This event is only sent when all previously created editor instances have already been disposed of. After this event, the program will not receive any more Editor Manager events. The event type is currently identical in contents to an X PropertyNotify event.

ENewInstance

```
typedef struct {
     XPropertyEvent   xproperty;       /* event that caused this action */
     Atom             instance;        /* new instance property atom */
     char             *variant;        /* variant string */
     char             *spec;           /* editor specification string */
     } ENewInstanceEvent;
```

Signals the external editor program to create a new editor instance. The instance atom, by which the new instance will henceforth be identified is returned in the instance field. The variant and spec fields contain the editor variant and the editor specification strings, respectively. These strings must be freed when they are no longer required.

After receiving an ENewInstanceEvent the editor program must respond by calling either EConfirmInstance or EDenyInstance, described below. A new instance should always be created with an initial value as specified by the default in the editor specification. The proper initial value, if there is one, will be sent through an additional forthcoming event.

EDisposeInstance

```
typedef struct {
     XPropertyEvent   xproperty;       /* event that caused this action */
     Atom             instance;        /* instance to dispose */
     void             *refCon;         /* instance reference constant */
     } EDisposeInstanceEvent;
```

Signals the editor program to dispose of the specified editor instance. The refCon field contains its reference constant. After this, the program never again receives an event concerning that instance.

At this point, the window in which the editor was created has already been destroyed.

EGetValue

```
typedef struct {
     XPropertyEvent   xproperty;       /* event that caused this action */
     Atom             instance;        /* instance of which value is req'd */
     void             *refCon;         /* instance reference constant */
     } EGetValueEvent;
```

Returned whenever the Editor Manager needs to know the current value of the specified instance. The refCon field contains the instance's reference constant. After receiving such an event, a program should respond by calling EReturnValue, described below.

ESetValue

```
typedef struct {
     XPropertyEvent   xproperty;       /* event that caused this action */
     Atom             instance;        /* instance of which to change value */
     void             *refCon;         /* instance reference constant */
     char             *value;          /* new instance value */
     } ESetValueEvent;
```

An editor program receives an ESetValueEvent to change one of its instance's value. The editor should either redraw the instance immediately to reflect the new value, or generate an Expose event on itself so that it is redrawn through normal event processing.

The instance field contains the appropriate instance atom, and the refCon field the reference constant. The string containing the instance's new value is stored in the value field. This string is an X resource and must be deallocated using the XFree routine when the program is done with it.

ROUTINES

The Editor Manager does not need to be initialized, but the X client must have been set up to receive PropertyNotify events from the root window in order for it to function correctly.

The Editor Manager interface is exceedingly straightforward—new editors are created by calling NewEditor, which returns a new editor reference number. While editors are in existence, their values can be obtained or changed by calling GetEditorValue and SetEditorValue. When the caller is done with the editor it should call DisposeEditor.

```
short NewEditor(Layout *layout, char file[], char variant[], char
        spec[]);
```

Creates and lays out a new editor instance and returns an editor reference number which is used to refer to the instance in later calls, or zero if the instance could not be created.

Layout specifies in which layout the editor is to appear.

File is the file name of an external editor, or NULL if an internal editor should be used. If necessary, a new editor process is started to create the instance.

Variant is the editor variant. For a command editor this is a full command name (the concatenation of all the subcommand step names), and for a parameter editor a type name. Spec is the parameter or command specification as returned by the Command Manager.

The editor's value is initially set to the parameter default, as given in the parameter specification.

```
void DisposeEditor(short ref);
```

Disposes of the specified editor instance, releasing all resources allocated for it. After calling this routine, the instance should definitely not be referenced again.

If the editor is an the last external editor instance of an editor process, the process is also terminated.

```
void SetEditorValue(short ref, char *value);
```

Sets the specified instance's new current value. The instance is automatically redrawn to reflect the new setting.

Value is a character string specifying the new value. The format of this string depends on the type of editor, as described in the ‘Editor specification’ section, above. The actual new instance setting when the format of this string is incorrect, is editor-specific.

```
char* GetEditorValue(short ref);
```

Returns a new string showing the current value of the specified editor instance, or NULL if there was an error.

```
void EditorWindow(Window panelWindow);
```

This routine is the only blemish on the symmetry of the Editor Manager interface. After all the editors in a particular dialog have been laid out and the layout is closed, the external editors need to know the newly created panel window. This is due to a quirk in the Xr panel mechanism—if another toolkit is used in the future, this routine can probably be removed.

Protocol interface

All external editor programs should use these routines rather than communicating with the Editor Manager directly through the process and instance properties. Changes or bug fixes to the low-level protocol will then automatically be available to the editors.

The module is contained in the extedit .c, .h and .o files. It uses the X Window intrinsics library, but no toolkit. It requires that the editor program declare and initialize two global variables named display and screen before using any of the routines. Display is returned by XOpenDisplay, and the screen can be obtained either from the X intrinsics DefaultScreen macro, a command line option or an .Xdefaults variable[3].

```
Boolean EOpenLink(char *propName);
```

Prepares the process property communication with the Editor Manager. PropName is the name of the property atom, as passed to the editor program by the Editor Manager in the argv[1] parameter to the program's main() routine. The return value indicates whether the routine was successful. If it wasn't, communication with the Editor Manager could not be established and the program should terminate.

Call this routine once only when the program has determined that it can run, after initializing the X Window library but before using any of the other extedit routines.

The routine obtains the process property atom from the specified property name and signals to the Editor Manager that process creation was successful.

```
void ECloseLink();
```

Reverses the operation of EOpenLink routine by terminating communication with the Editor Manager. Call this routine before terminating the editor process and before closing the connection with the X server.

Currently this routine does nothing because there is no actual connection which needs to be closed, but it has been incorporated into the routine interface to support future enhancements to the extedit module.

```
void EEvent(XEvent *event);
```

Examines the specified event to see if it applies to the Editor Manager communication. If it does, it processes the event accordingly and modifies the event record as appropriate. After calling this routine, the program can examine the event type returned and branch accordingly, just as with ordinary X events.

This routine *must* be called after any type of X event has been removed from the queue. Otherwise, the interface module may miss a crucial event and be left behind at an earlier stage in the protocol. However, if the editor program only examines the event without actually removing it from the queue (using one of the XCheckEvent routines), EEvent should *not* be called because the event would then be registered twice (which is just as bad as not getting the event at all). There is no facility for re-sending data after a certain period has elapsed, because there is really no reason why a communication should be lost.

Specifically, the routine looks for PropertyNotify events, which at different stages in the protocol can mean different things. Refer to the “Editor Manager events” section for more details. In future, the routine may also need to look for other X event types.

```
Boolean EConfirmInstance(Atom instance, Window *window, short *x,
         short *y, void *refCon, short width, short height);
```

After receiving an ENewInstanceEvent, causes the Editor Manager to be informed that the required instance can in fact be created by the editor program.

Returns TRUE if successful. If so, the program should create and start handling events for the new instance. If it returns FALSE, there was an error. This should really never occur, but if it does the

program should consider the new instance as not created but continue running, because any other running instances may still be able to function correctly.

Atom is the new editor instance atom, as returned in the instance field of the ENewInstanceEvent structure.

Window, x and y are pointers to where the instance's window ID and the location of its upper left corner will be stored upon return of the routine. Currently, a new mapped window is always created for the instance, and its upper left point is always located at (0, 0) within the specified window, but this may change in the future.

RefCon specifies a reference constant to be associated with the new instance. This constant, which is declared as a pointer to void but never accessed by extedit, is returned to the program whenever an event occurs concerning this instance.

Width and height should be set to the new instance's required width and height. This information is used by TRIP to find space between any other editors in the dialog window.

```
void EDenyInstance(Atom instance);
```

After receiving an ENewInstanceEvent, causes the Editor Manager to be informed that the required instance will not be created by the editor program. After using this routine, the editor program is never again contacted with regard to this new instance.

As mentioned before this routine should be used only sparingly, and probably only in instances in which the program cannot obtain the necessary resources to support the new instance.

```
void EReturnValue(Atom instance, char *value);
```

Call this routine in response to an EGetValue event to report an instance's current value. Instance is the property atom as given in the event's instance field, and value a string containing the current value. The string remains the property of the editor program, and should be disposed of by it when appropriate.

```
void* EGetRefCon(Atom instance);
```

Returns the reference constant of the specified editor instance, or NULL if the instance doesn't exist.

```
void ESetRefCon(Atom instance, void* refCon);
```

Sets the reference constant of the specified editor instance, overriding any previous value. If the specified instance doesn't exist, this routine does nothing.

FUTURE IMPROVEMENTS

As mentioned, the Editor Manager could maintain an editor process *latency policy* by allowing some editor processes to continue to exist while they are not supporting any instances. There would then be a substantial benefit if the process were indeed called upon to create new instances in the future. The process should put itself in a dormant state by disabling all non-critical events, to avoid having itself made runnable by the operating system scheduler unless absolutely necessary. The drain on system resources might not be that great as the operating system would probably swap an accordingly blocked process out pretty quickly. This statement has not been tested quantitatively.

4 Dialog Manager

And they said that in our time
All that's good will fall from grace
Even saints would turn their face
In our time

Hymn, Ultravox [Quartet]

The Dialog Manager is the interface between the yet to be implemented parts of TRIP and the editor system described in this document. (This will probably mean the Structure Manager, which is assumed for conversational purposes.) It offers the highest level of abstraction of a command editor in terms of other lower-level constructs such as editors, layouts and regions. An unlimited number of dialogs may be on the screen at any time.

In the context of this document, a **dialog** is a window containing all the necessary editors and associated control structures to specify the parameters to commands required to drive the simulator system:

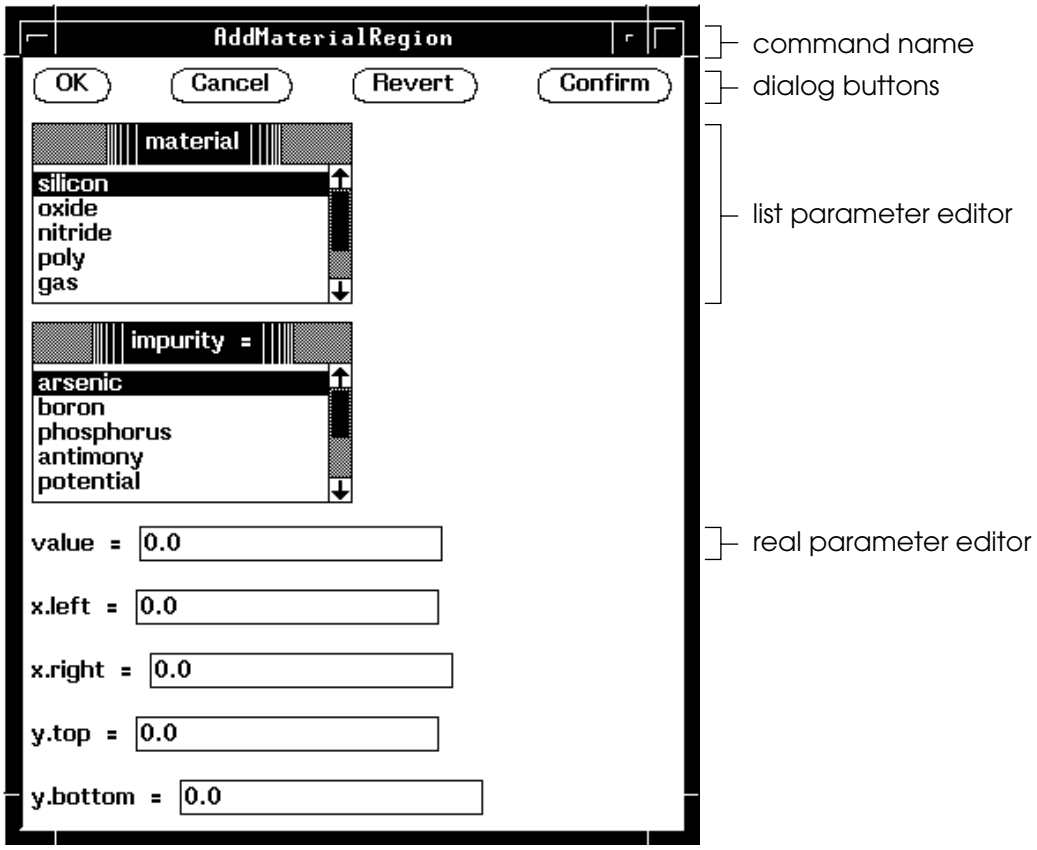

Figure 4-1. A dialog

The Dialog Manager uses data derived from the *command/dialog description file* [8, ch. 4] by the Command Manager to determine the parameters and parameter types which constitute a particular command. For example, the dialog in the figure was created by the Dialog Manager according to the following CDD file specifications for the TRENDY 'Add-Material-Region' command:

```
(Add
     (Material
          (Region
               <material : 'silicon, oxide, nitride, poly, gas, ambient,
                    back' : 'silicon'>
               <impurity = : 'arsenic, boron, phosphorus, antimony,
                    potential, electrons, holes, vacancy, interstitial' :
                    'arsenic' >
               <value = : posreal : 0.0>
               <x.left = : real : 0.0>
               <x.right = : real : 0.0>
               <y.left = : real : 0.0>
               <y.right = : real : 0.0>
               )
          )
     )
```

METAVALUES

Warning: The metavalue implementation was not complete at the time of this writing. Because creation of the Phase II software continued after completion of this document, there may be discrepancies with the implementation described here.

In the specification of the syntax of the CDD file, brackets can be used to indicate that a parameter, or a group of parameters may be specified an undetermined number of times:

`< … > < … >`	Specify parameters exactly once (*compulsion*)
`[ < … > < … > ]`	Specify parameters either once or not at all (*option*)
`{ < … > < … > }`	Specify parameters any number of times (*repetition*)

These constructs make the consistent generation of a dialog much harder. It has been suggested to me that the user could be prompted to enter the number of times he or she wishes to specify the parameter before the dialog is opened, and create a dialog according to this value—this, however, does not agree well with the premise that a dialog should be interactive and forgiving, allowing the user to make and correct mistakes at will. Also, this method could result in the creation of very large dialogs with a large number of identical editors in which to enter values, not a very appealing prospect.

Instead a solution was required which consistently results in the creation of an identical dialog for a particular command (thereby promoting familiarity to the user), does not create gigantic dialogs, but still offers a maximum of flexibility, allowing a user to change the parameter values and the number of values to the full extent of the specification, and in fact takes it much further. This has been achieved through the introduction of *metavalues*. The next subsections give a somewhat abstract description of metavalues—they will likely be more comprehensible on a second reading, after sampling the practical aspects of their appearance in the section on “Metavalues in dialogs”.

Definitions

A **metavalue** is, in its most general definition, a poly-dimensional space of editor values or other metavalues. For instance, a repetition represents a single metavalue consisting of an undetermined number of editor values or other metavalues. With this definition it seems obvious also to state that the value of a dialog itself is a metavalue of all its constituent parameter's values. A single parameter delimited by angle brackets can have just a single value which is, of course, a string returned by the editor. To emphasize this aspect of their relation with metavalues, single values returned by editors are also called **simple values**. Where the distinction is irrelevant, metavalues and simple values are in this chapter together referred to just as values.

Because the bracket constructs can surround more than one specification, we speak of the **tuple** of the metavalue as the number of component values. All three of the metavalues in the above examples are 2-tuples because they contain two separate parameter specifications. Even if one of the compulsions was

replaced by one of the other two types we would still have a 2-tuple. A dialog metavalue is an *n*-tuple, depending on (but not necessarily equal to) the number of parameters to the command.

The number of times that the parameter is actually specified by the user in the metavalue is called its **valence**. For example, the valence of a compulsion is always equal to one because it must always be specified exactly once. An option may have valence zero or one, and a repetition can have any integer valence. Simple values have neither a tuple size nor valence. The following figure shows some representations of metavalues with different tuple sizes and valences:

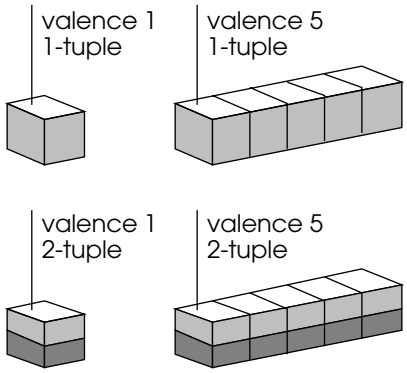

Figure 4-2. Rank-1 metavalues

The last term for classifying metavalues is its **rank**. The rank of a metavalue is the sum of the ranks of each of its constituent metavalues, plus one, where a simple value has rank 0. Higher-rank values are always proper metavalues. An example of a rank-1 metavalue is the 7-tuple dialog metavalue in figure 4-1. The rank of the compulsion is zero and that of the other two is one. (In other words, a compulsion by itself is not a metavalue but a simple value.) If one or both of the compulsions were surrounded by an additional level of option or repetition, the respective ranks would increase by one.

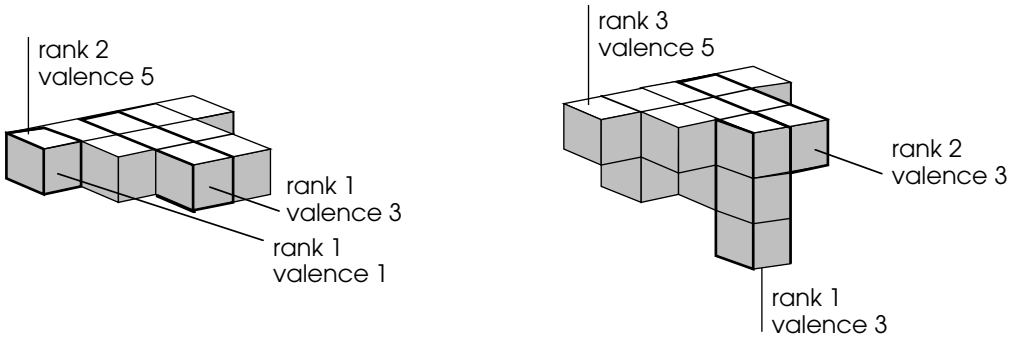

Figure 4-3. Higher-rank metavalues (1-tuples)

Metavalues with rank greater than 3 are difficult to picture. Although their usefulness may seem questionable at this time, no limit is imposed on their definition.

Each of the *n*-tuple cube shapes represents a metavalue **element**. The difference between an element and a simple value is that an element is a tuple:

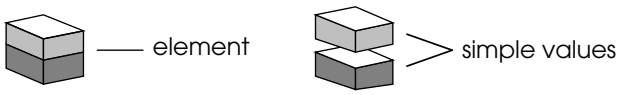

Figure 4-4. Elements and simple values

The important concept of the **active element** is used in the section on using dialogs. There is always exactly one element active, and that is the one whose simple values are currently visible and editable. Using dedicated field editors in the dialog to select which element is active, each of the elements in turn and thus the entire metavalue can be edited by the user. As metavalues may be thought of as

hypervolumes embedded in an *n*-dimensional value space, each of these special field editors may be said to specify a **coordinate** in this space.

Although the use of such exotic and unconventional terms may perhaps seem somewhat extravagant at first, they will likely make a more effective and durable impression on the reader's mind than something like ‘number’ or ’limit.’ These have already been overused, and unambiguous descriptions are helpful in explaining the concepts involved here.

Metavalue specification

The current CDD file command specification syntax defines a subset of the general metavalue. In particular, it allows the specification of some rank one nonzero-tuple metavalues. A new CDD file syntax incorporating general metavalue parameter definitions as well as other proposed enhancements is given in the Command Manager chapter.

If they are necessary, it should be very easy to implement other metavalue constructs than the ones given here. For example, another form of repetition could be defined which does not allow zero-valence metavalues, or one with a fixed valence that permits only coordinate selection from a user-specified list. Such a coordinate selector could be specified with an existing list type. The need for such constructs will in part be determined by the simulator for which TRIP is used.

Implementation

Metavalues are, not surprisingly, implemented as pointer trees. The basic building block is the following structure:

```
typedef struct _Meta {
     short            tuple,          /* tuple size */
                      allocTuple;     /* allocated tuple size */
     short            valence,        /* number of tuples */
                      allocValence,   /* allocated number of tuples */
                      active;         /* active tuple */
     Metaeditor       *editor;        /* editor array */
     struct _Meta     **value;        /* subvolume metavalues or simple */
     } Metavalue;
```

Tuple specifies the tuple size of the metavalue. AllocTuple is its actually allocated tuple size. The Dialog Manager grows metavalue tuples in chunks specified by the constant tupleAllocClump, to cut down on memory reallocation requests and fragmentation problems.

Similarly, valence specifies the metavalue valence and allocValence the actually allocated valence. Metavalue valences grow in chunks specfied by the constant valenceAllocClump. Active indicates the coordinate of the active element.

Editor is a pointer to an array of the following structures, one for each value in the tuple:

```
typedef struct {
     short            ref;            /* value editor */
     xrEditor         *coord;         /* coordinate editor */
     } Metaeditor;
```

If the value is a simple value, ref is an Editor Manager editor reference number for the simple value editor and coord is NULL. Otherwise the value is another metavalue, coord is the Xr editor instance controlling the valence coordinate, and ref is zero. Recall that actual editor reference numbers are always nonzero.

The last field in the Metavalue structure is a pointer to an array of other metavalues or simple values.

The type of field editor used to specify valence coordinates depends on the particular construct involved. For options, an Xr check box editor (like the boolean variant internal editor) will do nicely. For the repetition it will probably be necessary to write a new Xr editor type, like the one suggested and shown in figure 4-5, below. Metavalue constructs should be surrounded by a rectangle by placing them in sublayouts and then using an Xr groupbox editor.

Upon careful consideration it may appear there is some redundancy in the specification, because the editor is respecified for each element in the valence. Although it will often be the case that the editor will be the same regardless of the valence, allowing different editors to be used for each element could be very useful if, for instance, the individual elements represent command options which require different types of parameters, or parameters with different upper and lower bounds. Furthermore, metavalue ranks, tuple sizes and valences will practically always be very small so as to minimize any perceived inefficiencies.

If having separate editors for what amounts to as being the same command parameter in different options is objectionable, one solution might be to introduce 'virtual editors,' which would have different reference numbers, and so appear different from the Dialog Manager's point of view, would allow different attributes (such as their bounds) to be specified for them, but would use the same editor in the dialog. Of course, the CDD file constructs for these facilities have not yet been designed. But the metavalue model supports them all.

Flat representation

Although the metavalue representation described above is well-suited for the random access required while a dialog is in existence, or for the generation of simulator input files by the Structure Manager, the pointer tree is impossible to store in files. For this purpose an alternative 'flat metavalue' representation is defined which can be conveniently stored and converted to and from the regular metavalue pointer tree.

Although it can be used on any type of metavalue, the flat representation will normally be needed only for long-term storage of dialog metavalues in Structure Manager files. This would allow users of TRIP to save current work on a simulation file and reopen it to resume editing at a later time.

The flat representation is essentially a one-dimensional 'projection' of the value space of the metavalue. No information on valence limitations or tuple sizes is retained. This information is already specified in the CDD file. Therefore, it is important that flat metavalues always be correctly matched with their respective simulator commands so that the pointer tree can be properly rebuilt.

The format of the flat representation is defined as a recursive traversal of the value space. It should not be accessed directly, but only through Dialog Manager routines. The following C-style declaration is illustrational only:

```
union _FlatMetavalue {
      struct {                                  /* metavalue */
            short            valence;           /* metavalue valence */
            struct _FlatMetavalue metavalue[tuple][valence];
            }
      char       simple_value[][tuple];       /* simple value */
      } FlatMetavalue;
```

Information about the structure of the command is used to determine whether an array of other metavalues follows, or a simple value represented by a null-terminated C string.

For example, the dialog in figure 4-5 might have the following flat metavalue:

```
silicon arsenic 0.0 2 0.0 0.0 2.0 2.0
```

USING THE DIALOG MANAGER

After initializing the Dialog Manager, it may be instructed to create a dialog for a particular command by means of a single call. Again, there is no inherent limit to the number of dialogs that may be in existence at any time. After this call, the handling of the dialog proceeds completely asynchronously from the rest of the system, and independently as well, except for the passing of X events to the Dialog Manager. The user may at this point start editing the parameters in the dialog.

These changes, however, are not communicated back directly. Instead, the editors themselves keep track of the current settings, which allows the user to experiment with the various settings before committing them. Only when the user indicates he or she is satisfied with the settings (by clicking an appropriate

button in the dialog), or when the Structure Manager explicitly requests it, is the dialog metavalue changed.

The dialog metavalue contains the selected values of the command parameters in the pointer tree, which can be accessed by the Structure Manager. When the dialog is disposed, the metavalue can be converted into a flat metavalue which should be saved by the Structure Manager as the command value representation. Whenever the user wishes to reopen a command dialog to change the parameter settings of a command, the block can be unflattened by the Dialog Manager which will then use those values as initial values for the command parameters. Whenever the user introduces new simulator commands in the Structure Manager, no value is passed and the Dialog Manager uses default values which were specified in the CDD file.

The dialog is removed when the user indicates he has finished with it by clicking a button in the dialog. Note that this does not necessarily entail communication of the most current dialog settings to the Structure Manager. The dialog can also be closed under program control by calling a Dialog Manager routine.

When the Dialog Manager is shut down all remaining dialogs are automatically closed.

Commands selected by a user from a menu hierarchy which should result in the creation of a command dialog are specified to the appropriate Dialog Manager routine by means of the Command Manager PathCD structure, rather than (as originally intended) by a character string containing a literal representation of the path traced through the menu tree. This has prompted some changes to be made to the Command Manager. More information on this and the rationale for the change, as well as other changes introduced can be found in the following chapter on the changes to the Command Manager.

While the Dialog Manager is running it keeps a linked list of open dialogs, and which, if any, of the dialogs is **active**. When running under a window manager, the active window is the one currently receiving keyboard input. How active windows look different from others, and the way in which windows may be made active depends on the window manager used. The window manager which currently runs on the ICE's HP system (hpwm) shows active windows by highlighting its border, and windows may be made active by clicking in this border.

Dialog creation algorithm

There are many different ways in which the Dialog Manager may possibly create a dialog for a particular command. It uses the following algorithm:

- First, it tries to find an external command editor for any command in the command hierarchy starting with the original command, with the original command as the editor variant. Using the example CDD specification shown above, since the selected command was ‘Add-Material-Region’ the Dialog Manager attempts to create variant ‘AddMaterialRegion’ command editors for, in turn, ‘Add-Material-Region’, ‘Add-Material’ and ‘Add’.

Note: As of now, the Command Manager only allows the definition of command editors at the top level of the command hierarchy. Searching for command editors from the lowest level was incorporated in the Dialog Manager in anticipation of an enhanced future version of the Command Manager.

If no such editors exist, or if the attempts to create them were unsuccessful, the Dialog Manager will instead create a dialog from individual parameter editors. For each of the parameter types in the command specification,

- It tries to find an external parameter editor for any type in the type hierarchy, starting with the original parameter type, with the original type as the variant. For example, if the parameter type is ‘temp’ the Dialog Manager attempts to create variant ‘temp’ parameter editors for, in turn, ‘temp’, ‘posreal’ and ‘real’.

Note: As of now, the Command Manager does not allow the specification of external parameter editors. Searching for parameter editors along the hierarchy of the type was also incorporated in anticipation of future Command Manager versions.

Note the analogy between parameter and command editor hierarchies. If this fails for a particular parameter,

- It tries to open the internal editor for the parameter type's primitive type. In the preceding example, it will open the 'real' internal editor. This request can fail only due to extrinsic factors (such as a lack of memory).

If the algorithm seems somewhat elaborate, it can actually be performed quickly because of the improved Command Manager interface. In fact, practically all dialogs will be created according to the last step in the algorithm. Also, the number of parameters to a command and the hierarchy depth of commands and parameters will not normally be prohibitive.

I briefly considered, from a symmetrical standpoint, to also insert a test for the presence of an internal command editor in the dialog creation editor. Although the Editor Manager would certainly be able to support such a concept, I did not incorporate this capability because the TRIP system should be independent of the particular simulator system it is being used for, and it is very difficult to conceive of a general-purpose command editor which could be used by many different simulators.

USING DIALOGS

This section is concerned with dialog interaction from the user's point of view. Refer also to figure 4-1 at the beginning of this chapter.

When the TRIP software is completed, the user will be able to select a simulator command step from a hierarchical pop-up menu. Simulator commands appear as icons in a Structure Manager window. Double-clicking or a similar action on one of these icons will cause the respective command dialog to appear.

Although it should be reasonably clear how to interact with the editors in a dialog, (the editors are meant to be designed for this!) express information may be found in [1, 2]. Details on working under the Hewlett-Packard or Motif window managers may be found in [11].

Dialog buttons

Each dialog has a standard set of **dialog buttons** at the top which are used to control its interaction with the rest of the TRIP system. The particular positioning of these buttons may be changed in other versions of the Dialog Manager.

To avoid confusion, care should be taken that they always appear to the user in a consistent location. This is currently ensured because they are always the first to be placed in the dialog layout. In general, any changes which affect the user interface should be made with the greatest care and consideration for the user. An indication of how such an interaction policy might be implemented can be found in the *Macintosh User Interface Guidelines* [7, vol. I ch. 2, vol. IV ch. 1, vol. V ch. 1]. Many of the principles delineated here carry over directly to other systems as well. It is my opinion that the importance of a well-designed, consistent user interface cannot easily be overestimated.

Changes made by the user to the values in the dialog are stored locally with the editors and do not have any immediate effect on the rest of the system. This allows the user to freely experiment with the dialog without necessarily affecting previous values. The dialog buttons control the relationship of the values in the editor with the Structure Manager's idea of the values:

- **OK.** Clicking in this button causes the command's previous values to be replaced by the current values in the dialog, and the dialog to be closed.

- **Cancel.** Simply disposes the dialog. Changes made to the dialog since opening or the last 'confirm' operation are not applied.

- **Confirm.** Commits the current dialog settings without closing the dialog, allowing further editing in the dialog.

- **Revert.** The dialog reverts to the settings it had either when it was first created, or at the time of the last ‘confirm’ operation, whichever occurred most recently.

Metavalues in dialogs

The following figure shows how a metavalue dialog might be generated in response to the TRENDY ‘Add-Material-Polygon’ command specification:

```
(Add
      (Material
            (Polygon
                  <material : 'silicon, oxide, nitride, poly, gas, ambient,
                        back' : 'silicon'>
                  <impurity = : 'arsenic, boron, phosphorus, antimony,
                        potential, electrons, holes, vacancy, interstitial' :
                        'arsenic' >
                  <value = : posreal : 0.0>
                  { <x = : real : 0.0> <y = : real : 0.0> }
                  )
            )
      )
```

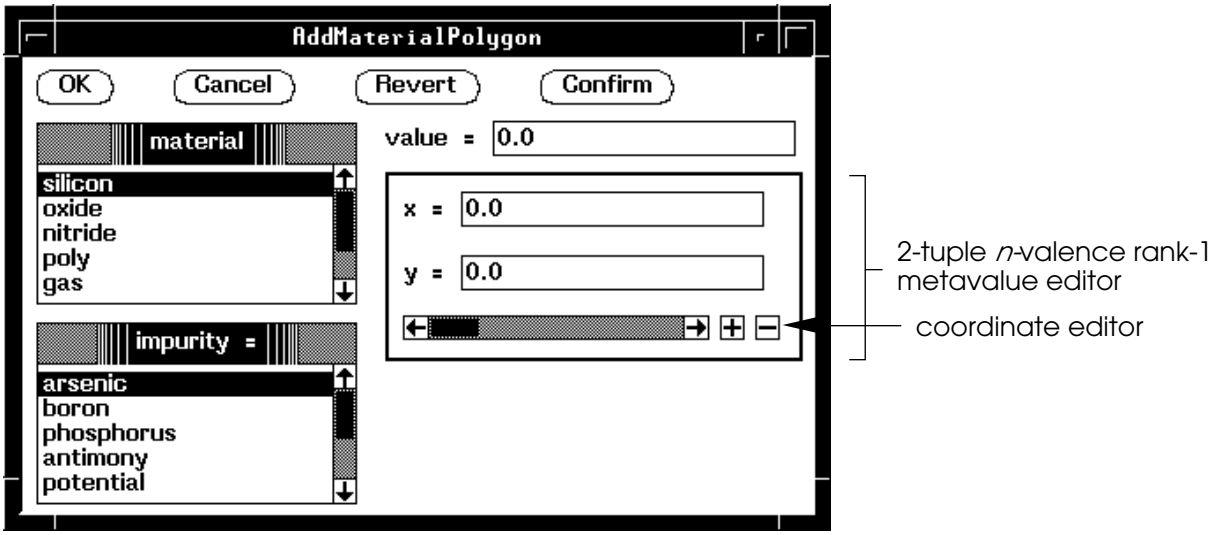

Figure 4-5. Simulated metavalue dialog

In this simulated dialog, the repetition construct parameters appear together in a rectangle. The value editors in the rectangle show the values of the active element and can be edited as expected. The special valence coordinate editor at the bottom allows the active element to be selected, as well as the valence to be changed. The scroll bar-part selects the active valence coordinate. The gadget (in Xr terminology) marked ‘+’ inserts a new element into the metavalue at the active coordinate, and the ‘–’ button deletes one. In this way, metavalues can be edited to the full extent of the CDD specification.

Since the implementation of metavalues should be completely hierarchical, other more complex examples can be easily extrapolated from the one shown here.

Because of the way metavalues are implemented, changing the active element also confirms the previous element's value.

ROUTINES

The Dialog Manager routines are spread over two sets of files. The dialog management routines are contained in the ‘dialog’ files. These routines will need to be used by the Structure Manager. Internally, the dialog routines use the metavalue routines in the second set of files, ‘meta’.

Warning: Because writing of the Phase II software continued after completion of this document, there will very likely be discrepancies between the set and interfaces of the Dialog Manager routines. At the time of this writing, the implementation of metavalues had not yet been completed. If you are planning to use the routines or modify their implementation, be sure to first check the appropriate source files.

Dialog management

```
void InitDialog(int argc, char *argv[], void (*closeDlgProc)(), void
         (*valueDlgProc)());
```

Initializes the Dialog Manager for use. Call this routine once at the start of the program, before using any of the other Dialog Manager routines.

Argc and argv should be set to the argument count and argument value list parameters to the main() routine of the TRIP program. Window manager properties can be set to these values which, when a window manager is running which uses them, will allow the TRIP application to be restarted or a duplicate to be launched without having to retype the command line that originally started the application.

CloseDlgProc and valueDlgProc are both pointers to functions you supply which are called by the Dialog Manager whenever a particular event occurs, or NULL if you don't want a routine to be called. The routines should be of the form

```
MyCloseDialogProc(Dialog *dialog);
MyValueDialogProc(Dialog *dialog);
```

The closeDlgProc is called whenever a dialog is about to be closed, but before it has actually happened. This allows any additional actions to be performed, such as keeping track of how many and which dialogs are currently in existence. A dialog can be closed by the user by clicking in an appropriate dialog button, or under program control by using either ShutDownDialog or DisposeDialog.

ValueDlgProc is called when a dialog value is committed. At this time the dialog value will have changed already. This kind of ‘event’ can be used to keep track of which dialogs have changed since they were last saved. This too can occur unpredictably, at the user's demand by clicking in one of the dialog buttons or induced through a call to CommitDialog.

InitDialog also selects PropertyNotify events on the root window of the display and screen on which TRIP is currently running, which are needed by the Editor Manager.

```
void ShutDownDialog(Boolean saveChanges);
```

Terminates all current activities of the Dialog Manager, including closing any dialogs which may still be in existence at this time.

If there are indeed any dialogs, and if any of these have changed value since they were last saved, the user should already have been queried as to whether these changes should be saved. passed the answer in saveChanges. If this is TRUE, the Dialog Manager will update the dialog values and generate ‘value events’ for every changed dialog through the valueDlgProc passed to InitDialog. The closeDlgProc is also called for each dialog.

```
Dialog *NewDialog(PathCD *cmdPath);
```

Creates and displays a new command dialog on the screen and returns a pointer to the associated Dialog structure, or NULL if the dialog could not be created.

CmdPath specifies the command which is to be dialoged. This parameter should be naturally available by using the available Command Manager routines for manipulating the PathCD data structure while the user makes the menu selection.

NewDialog initially creates an unmapped window, registers it with the Xr toolkit and sets the standard window manager properties for it, which include the window's icon and window and icon titles. The window title is set to the name of the command being dialoged. It opens a layout for adding editors by the Editor Manager. When this is done, it closes the layout and maps the window, causing it and the editors to be displayed. Note that if there is no window manager running, there is no explicit window title so that it may be difficult to tell which command a dialog belongs to when there are many dialogs on the screen.

```
void DisposeDialog(Dialog *dialog);
```

Disposes of the specified dialog without committing any recent changes which may have been made. A 'dialog close' event will be reported through the closeDlgProc passed to InitDialog. There should probably be no reason to explicitly dispose of dialogs, except when the program terminates. In that case, use ShutDownDialog instead to dispose of all the dialogs at once.

```
Metavalue* ValueDialog(Dialog *dialog);
```

Returns the specified dialog's metavalue. This is always the same data structure which is used by the Dialog Manager, and will therefore change value automatically whenever they are committed.

```
void CommitDialog(Dialog *dialog);
```

Causes the dialog metavalue to be updated to reflect the current settings of the dialog editors. There should probably be no reason to explicitly commit new dialog settings, and the user may not expect to see what may be 'experimental' settings suddenly incorporated in the real simulator input.

```
Boolean DialogEvent(XEvent *event);
```

Examines the event and returns TRUE if it 'belongs to' the Dialog Manager. If so, DialogEvent will perform all the necessary event processing, and the event may be ignored by the caller. If the routine returns FALSE, the event was not a Dialog Manager event and should be processed normally by the calling routine.

Call this routine whenever an X event is removed from the queue, from either X or Xr event-gathering routines. It is particularly important that DialogEvent receive all FocusIn/FocusOut events on dialog windows so that it knows when one becomes active.

```
void DialogTask();
```

This routine takes care of dialog input event processing for the Dialog Manager. You should call this each time through your event loop if you want active dialogs to respond to user input, such as mouse clicks and keypresses.

Metavalue routines

```
void* DeflateMeta(Metavalue value);
```

Returns a flat representation of the specified metavalue, or NULL if there was an error.

```
Boolean InflateMeta(Metavalue value, void* flat);
```

Sets the metavalue to the values as specified in the given flat representation. The result indicates whether the operation was successful.

FUTURE IMPROVEMENTS

Text edit *breaks,* caused by pressing the ‘tab’ key when using a text edit Xr field editor, should be used to skip the text insertion cursor to the next text editor in the dialog. Currently, text edit breaks are detected but ignored.

Type checking and subtype range checking have not yet been implemented due to lack of time. Also, it is possible that the extension of the Command Manager with a dynamic attribute definition capability will introduce substantial changes to the way in which subtypes are defined in terms of parent type attributes, such as upper and lower bound.

In future, the use of a different toolkit should be seriously considered. Although the Xr toolkit suffices for the current implementation, some of its limitations are already becoming clear. The use of multiple panels and in particular, the correct handling of exposure events in such situations is not at all obvious. Even the Xr example programs (such as *RB*) do not correctly support multiple panels where they use them. Also, the only available documentation on Xr [1] was inadequate. For one thing, it is extremely difficult to determine how long particular structures need to remain in existence, and whether they are freed by Xr or if this is the caller's responsibility. And lastly, the fact that the Xr field editors have suddenly gained the characters “10” in their manual page titles is foreboding. It appears that the Xt toolkit may hold more promise for standardization and widespread use within the ‘X community.’

5 Command Manager

I've had my fun and now it's time
to serve your conscience overseas

—Orange Crush, R.E.M. [Green]

This chapter describes the version 1.1 changes and extensions to the Command Manager described in [8]. In addition to the intentional amendments made to the functionality, some bugs were also corrected. As far as it describes existing features of the Command Manager the material in this chapter supersedes the older material. The reader is assumed to be familiar with the appropriate chapters of that report.

The main purpose of the Command Manager is the parsing of the Command/Dialog Description file. The CDD file contains the specifications for the commands and their parameters of a particular application for which input is being generated. The Command Manager uses the information in this file to build an internal representation which may be randomly accessed by the other managers whenever they need this information.

The Dialog Manager uses Command Manager routines to dissect commands into their component parameters, to find out how certain data types are defined in terms of other data types, and to inquire whether a particular command or data type has a dedicated editor program associated with it.

SUMMARY OF CHANGES

• Printing of error and diagnostic messages was rerouted to stderr. Other than being standard UNIX practice, this is necessary to avoid confusing the simulator if ever a solution is realized in which TRIP command output is routed directly to the simulator through a pipe. This would be done by starting TRIP from the command line by something like

```
trip | trendy
```

• Instead of assigning the name “DefaultEditor” to any command editor which is not specified in the CDD file, NULL is used instead to indicate no external editor. This is more convenient and eliminates some failure modes. An explicit name was not needed because the ‘default editors’ do not run in a separate process.

• There is a slight typographical error in the formal grammar [8, p.19]. The line containing the definition for the “par spec” nonterminal should read:

```
<par spec> ::= { <par> { '{' { <par> } '}' | '[' { <par> } ']' } }
```

or, alternatively,

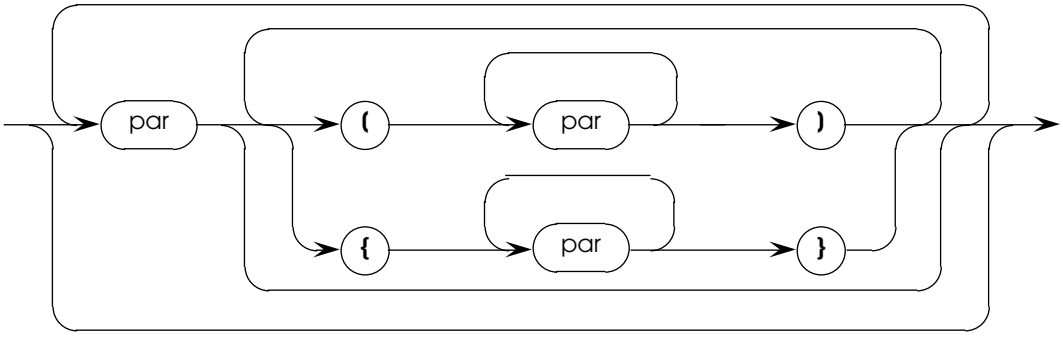

Figure 5-1. *par spec* nonterminal syntax flow diagram

to include both the sequential and optional parameter definition forms. Also, the “type” nonterminal should include a branch for the specification of quoted strings for list types:

```
<type> ::= <identifier> | ' <string> '
```

Note that single and not double quotes are used to delimit list type strings.

An alternative syntax for the specification of commands, which more fully exploits the capabilities of the new Dialog Manager is given later on.

• The nature of the parsing of the CDD file is that it translates a literal, one-dimensional representation of the command hierarchy and specification into a structured poly-dimensional pointer-tree abstraction which is more suitable for processing and examination. At the time of the writing of version 1.0 it could not be foreseen that the Dialog Manager would still need to be able to examine the structure and hierarchy of the command, so that communicating commands by means of their literal representations (e.g., “Add-Material-Region”) seemed sufficient .

However, in order to support some of the suggested enhancements of TRIP's capabilities (stressing the hierarchy of command definition), the Dialog Manager needed greater access to command structure information. The literal representation is unsuitable for this purpose, because each time command information needs to be accessed the command's name itself must be compared against existing names in the ‘command data base’ (as the case is, in a linear fashion).

Therefore, a mechanism which was already in place is now used to specify commands, namely the PathCD structure. This data type specifies the path from the root of the command hierarchy to the selected command by a list of ordinal numbers of branches which must be followed to reach it. Although it is certainly true, that to access command information it is still necessary to traverse the command tree, the effort involved here is practically negligible as there is no ‘searching’ or comparison involved in the traversal and the depth of command definition is bound to be extremely limited. An even more efficient solution, the use of simple pointers to the command information structures (e.g., CElement or OElement) was briefly considered but abandoned, as it is possible that the structure of the command definition tree may change significantly when further enhancements are introduced in a later version.

External editors

In the version 1.1 Command Manager the concept of external editors was generalized. Previously, specification of external command editors was possible only for the first level of command definition, e.g. for “Add” but not for “Add-Material-Region.” In the following discussion, note the symmetry between the handling of commands and parameters.

> **Note:** Unfortunately there was not enough time to rewrite the Command Manager to fully incorporate these changes—there is still no way to specify use of the new editor types in the CDD file. Furthermore, it was deemed best to defer any substantial changes to the CDD file syntax and semantics until after the expected additional enhancements have been completely specified. Specifically, enough was done so that the Dialog and Editor Managers could already take advantage of the new extensions and would not have to be modified when these are fully implemented. In effect, the Command Manager now only ‘pretends’ to support the new editor types.

In general, the particular operation of an editor will depend on subcommand ('option') selections. For instance, an editor for the "Add-Material-Region" command would probably differ greatly from one for "Add-Profile." Therefore, command editors may now be attached to any branch of the command, so that different editors can be attached to the aforementioned commands.

Conversely, it is reasonable to assume that editors for "Add-Material-Region" and "Add-Impurity-Region" might be very similar. In these cases it would be a waste to have to duplicate much of the code to write another editor which differs only in a few details. For this reason it is now possible to attach the *same* external editor to different commands. In fact, subcommands already *inherit* a parent command's editor. If necessary, the external editor can differentiate between its different instantiations through the editor variant it receives from the Editor Manager. For command editors, this variant is simply the full name of the command which invoked it, e.g. "AddMaterialRegion". A command editor may choose to refuse requests to handle certain commands.

Instead of writing editors to handle complete commands, it will also be extremely convenient to be able to write editors to handle only a particular parameter type. These parameter editors, which are identical in structure to command editors, are invoked whenever a particular data type is needed for a command parameter. For instance, it might be appealing to write an editor for temperature data which displays itself in the form of a thermometer.

Again, different types may make use of the same external editor. In fact, subtypes already inherit a parent type's editor. If necessary, the external editor can differentiate between its different instantiations through the editor variant specification it receives from the Editor Manager. For parameter editors, this variant is simply the name of the type which invoked it, e.g. "temp" for temperature type data values. A parameter editor may choose to refuse requests to handle certain types when it deems they are inappropriate for it.

CDD GRAMMAR

The functionality of the new Managers has been greatly enhanced from the original definition. The Command Manager, and specifically the CDD file, do not use these features yet. A new CDD file syntax for command specifications is proposed here which allows the user to fully exploit the new capabilities.

The part of the grammar dealing with single command specifications is, with one exception, a superset of the existing one—i.e., most of the specifications written under the existing syntax will be compatible with the new syntax. Command specifications are actually not parsed by the Command Manager but by the Dialog Manager at the command level, by the Editor Manager at the parameter level and by the individual editors at the level of individual parameter specification components. They are described here to keep the grammar in one place, and because users writing CDD files do not really care which part of TRIP parses what. The careful design has also made it possible to implement the new Managers so that they already support the new format, even though they currently never actually get to 'see' it. In other words, when the Command Manager parser is changed the Dialog Manager will automatically *immediately* support the enhancements.

The compatibility exception is the specification of list type parameter default values. The requirement that they must be delimited by single quotes, like the list type specifications themselves, has been removed in the proposed grammar. The quotes were not necessary anyway but introduced a special case which had to be detected and excepted to remove the quotes. This change could not already be implemented because it would have made the existing CDD files and Command Manager parser incompatible.

The new formal grammar for single command specifications is, like the original, given in Backus-Naur Form. Although grammars written using it are not always immediately obvious, they have the undenyable advantage of providing a consistent representation which has become widely popularized, first through its use in [16]. Because the CDD syntax itself uses some of the BNF metasymbols (square, curly and angle brackets) as terminal characters, they are surrounded in single quotes in these instances.

```
<command specification>     ::= <metaparameterlist>
<metaparameterlist>         ::= { <metaparameter> }
<metaparameter>             ::= <option> | <repetition> | <aggregation> |
                                <simple parameter>
<option>                    ::= '[' <metaparameterlist> ']'
<repetition>                ::= '{' <metaparameterlist> '}'
<aggregation>               ::= '(' <metaparameterlist> ')'
<simple parameter>          ::= '<' <label> : <type> : <value> '>'
<type>                      ::= <identifier> | <list type>
<list type>                 ::= ' <list item> { , <list item> } '
<list item>                 ::= <string>
<label>                     ::= <string>
<value>                     ::= <string>
```

The introduction of the <metaparameter> nonterminal is analogous to the metavalue. Note that there is no longer any restriction on the level of nesting (the rank) of the metaparameter grouping constructs, nor is it required any more that a variable-valence metaparameter always be preceded by a valence-one compulsion.

Note also the definition of the <type> nonterminal clearly shows the special status of list types. As noted at the end of the chapter, it should also be made possible to define list subtypes in terms of other list types, so that an identifier can also be associated with a list type.

An **option** is a construct of parameters which may be specified zero or once in a command. For instance, in the TRENDY grid definition command the ‘number’ parameter is optional, as the following fragment of CDD source adapted from [8] shows:

```
<direction : 'X-dir, Y-dir' : X-dir>
<position : real : 0.0>
<spacing : posreal : 0.0>
[ <number : posint : 1> ]
```

A **repetition** allows a set of parameters to be specified any desired number of times, as in the TRENDY Add Material Polygon command. Here, a polygon is defined as a repetition of *x*, *y*-coordinate pairs:

```
<material : 'silicon, oxide, nitride, poly, gas, ambient, back' : silicon>
<impurity : 'arsenic, boron, phosphorus, antimony, potential, electrons,
            holes, vacancy, interstitial' : arsenic>
<value : posreal : 0.0>
{ <x : real : 0.0> <y : real : 0.0> }
```

Finally, a new construct, **aggregation**, is introduced. It does not affect the parameter valence (the number of times that it may be specified) but instead indicates that a group of parameters is semantically related and should be kept together in a dialog, making use of the Layout Manager sublayout feature. Its value is demonstrated in the much-quoted Add Material Region command specification:

```
<material : 'silicon, oxide, nitride, poly, gas, ambient, back' : silicon>
<impurity : 'arsenic, boron, phosphorus, antimony, potential, electrons,
            holes, vacancy, interstitial' : arsenic>
<value : posreal : 0.0>
( <x.left : real : 0.0> <x.right : real : 0.0>
  <y.left : real : 0.0> <y.right : real : 0.0> )
```

We are thus ensured that the rectangle's coordinates appear together as a related group. The first two constructs are actually implied aggregators, causing their enclosed parameters to be grouped also.

ROUTINES

Following is a summary of the new or modified routines in the Command Manager. For the modified existing routines, a full function specification has been restated so that the reader need not also refer to the original definition.

Routines which are said to return “a new string” create a new string which must be deallocated by the caller when it has finished using them. Failure to do so will not only waste precious resources but will result in the buildup of large numbers of small objects and slow down the allocation process [9, vol. 1, p.

275]. Most of the version 1.0 Command Manager routines create and return new strings. On the other hand, routines which return "a Command Manager string" do *not* create a new object but return a reference to an existing string previously created by the Command Manager. These strings must not be changed in any way and must not be deallocated by the caller.

```
int GetParentCD(char *vname, char **parent);
```

Returns in *parent a pointer to a new character string containing the name of the parent of the type vname, or NULL if the specified type has no parent (i.e., it does not exist or it is one of the primitive types integer, real, string, Boolean, or a list type) or if an error occurred. Error codes returned by this function are unchanged from the previous version: 0 if there was no error, –1 if the specified type has no parent, and –2 in the case of a memory allocation error.

This routine was changed to return NULL in *parent if the parent doesn't exist rather than creating returning a zero-length string object. Returning an empty string is appropriate when the string needs to be printed but in type tree traversal the NULL result is more natural.

```
int GetSuperParentCD(char *vname, char **parent);
```

This routine is like the one above, except that instead of returning a new string containing the name of the direct parent of the type, it returns its superparent, which is always one of the primitive types, or NULL if there was an error. Primitive types are also defined to be their own superparent. The function returns 0 if there was no error, –1 if the specified type does not exist, and –2 in the case of a memory allocation error.

In the previous version of the routine, it was an error to ask for the superparent of a list type.

```
char *NextParameterSpec(char **cmdSpec);
```

Returns a new string containing the next parameter specification in the given command specification, or NULL if there are no more or if there was an error. The pointer *cmdSpec is changed to point to the first character following the end of the parameter specification just returned in the command specification.

This routine is used to convert a command specification into a series of constituent parameter specifications.

Command specifications are returned by the CommandSpec routine, below.

```
char *ParameterType(char *param);
```

Parses the given parameter specification and returns a new string containing the parameter type, or NULL if there was an error.

```
char *ParameterEditor(char *type);
```

Returns the Command Manager string containing the name of an external parameter editor for the specified type, or NULL if there is none. What it does not do is to return the name of a parent type's editor if it has one. In other words, the type hierarchy must be explicitly retraced in order to find a parameter editor at any level.

> **Note:** As there is currently no provision in the Command Manager for the specification of external parameter editors, this routine always returns NULL. One exception is the "temp" parameter type, for which it returns the same string as the name of an external editor. This is necessary for demonstrational and validational purposes.

```
short CommandDepth(PathCD *cmdPath);
```

Returns the 'definition depth' of the command specified by cmdPath. The first level of commands (i.e., those which would be selected from the first level in a menu) has a definition depth zero, and deeper lev-

els (also called *options*) have successively higher depths. If the path does not represent a valid command selection, the result of this routine is undefined.

This routine is used by the Dialog Manager in conjunction with the CommandEditor routine below, to retrace a command definition back up to the root of the hierarchy when looking for command editors.

```
char *CommandName(PathCD *cmdPath);
```

Returns a new string containing the full name of the command specified by cmdPath, or NULL if there was an error. A **full name** consists of the concatenation of all the subcommand names in the command's derivation hierarchy, e.g. “AddMaterialRegion”.

```
char *CommandSpec(PathCD *cmdPath);
```

Returns the Command Manager string containing the command specification of the command specified by cmdPath, or NULL if there was an error. The specified path must actually refer to a command and not a menu option with additional subcommands, or NULL will be returned. A command specification is the concatenation of all the specifications of the parameters to the command, i.e. the <par spec> nonterminal in the formal grammar. A command specification would therefore typically have the form of something like “<…> <…>”

```
char *CommandEditor(PathCD *cmdPath, short level);
```

Returns the Command Manager string containing the name of the external command editor for the specified level of the command derivation, or NULL if there is none or there was an error.

For example, if the selected command corresponding to cmdPath was “Add-Material-Region,”

```
CommandEditor(amrPath, 2);
CommandEditor(amrPath, 0);
```

will return the names of, respectively, external command editors for “AddMaterialRegion” and “Add.”

> **Note:** As there is currently no provision in the Command Manager for the specification of external command editors at any other than the first command level, this routine always returns NULL for nonzero levels

FUTURE IMPROVEMENTS

The following suggests some improvements for a future version of the Command Manager.

The practice of returning explicit error codes as function results is questionable, since it monopolizes an important fraction of the function communication ‘bandwidth’, most of the time merely to indicate that the function executed correctly. The actual functional result of the routine must then be communicated to the caller through some other, less obvious, way—for instance, through the use of a call-by-reference function parameter. On the other hand, if this *functional* result were indeed to be returned as the *function* result, it is practically always possible to indicate the fact that an error occurred by returning an unusual function result (such as a NULL pointer). The actual error code could then be accessed, for instance, in a designated global variable or through an error routine.

The current handling of types definitely needs also to be changed, in a similar fashion and for similar reasons as the handling of commands was. This has not already been done in the current version. Types could be specified and communicated using an integer number assigned by the Command Manager when the CDD file is parsed.

If the Command Manager were to assume the task of parsing the parameter specifications (i.e., the <par> and <type> nonterminals), efficiency of the Editor Manager could be improved. As the case is, the Editor Manager parses and needs to create many small objects each time, for instance, a list type is dialoged.

Such a redefinition of the handling of types should also allow a change in the way list types are used in the CDD file. Currently, a list type must be entirely restated each time it is used in a parameter specification. By allowing the definition of list types in a similar way as that of other types, i.e. in a separate statement outside of the command definition statements, this retyping would be avoided. The Editor Manager could also be made to run more efficiently because it would be working with a single, rather than many identical types.

An improvement to type handling should also disallow the redefinition of existing types or their attributes (such as the lower and upper bounds of the type), which might conceivably cause problems in some cases.

The parameter structuring capability in the proposed new grammar might conceivably improve the layout of certain command dialogs under some conditions, by keeping related parameters together in the dialog in a consistent manner.

If the CDD syntax is to be enhanced, it would very likely be worthwhile to completely rewrite the parser using the excellent UNIX parser-generator tools, lex and yacc.

Bibliography

[1] Hewlett-Packard, *Programming With the X Window System, HP 9000 Series 800 and 300 Computers*

[2] Hewlett-Packard, *Getting Started With the X Window System*

[3] *Xlib Programming Manual for Version 11,* ed. Adrian Nye, O'Reilly and Associates, Inc.

[4] *Xlib Reference Manual for Version 11,* ed. Adrian Nye, O'Reilly and Associates, Inc.

[5] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language,* second edition, Prentice Hall

[6] R. Remmers, *Spreading Resistance and SIMS Profiling of Silicon Structures,* 250-hour assignment report, July 1990

[7] Apple Computer, *Inside Macintosh,* Addison-Wesley, Reading, Mass., 1985

[8] P.F.A. Middelhoek, *Design and Partial Implementation of TRIP, a Flexible Input Processor for TRENDY,* 250-hour assignment report, October 1989

[9] Bell Telephone Laboratories, *UNIX Programmer's Manual,* Murray Hill, New Jersey

[10] Hewlett-Packard, *Using the X Window System, HP 9000 Series 300/800 Computers*, part number 98794-90001

[11] Hewlett-Packard, *A Beginner's Guide to the X Window System, HP 9000 Series 300/800 Computers*, part number 98794-90001

[12] Kemal Efe, *A Proposed Solution to the Problem of Levels in Error-Message Generation,* CACM (30) 11, November 1987, pp. 948

[13] Hewlett-Packard, *HP C/HP-UX Reference Manual,* part number 92453-90024

[14] S.R. Bourne, *The UNIX System,* Addison-Wesley, Reading, Mass., 1982

[15] James D. Foley, A. van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, Mass., 1984

[16] P. Naur, et al., *Report on the Algorithmic Language ALGOL 60,* CACM (3), 1960, pp. 299

Index