aXe

The User Guide for Release 6.0

Department of Computing Science
University of Newcastle upon Tyne
February 1994

Jim Wight <j.k.wight@newcastle.ac.uk>

Table of Contents

1 A beginner's guide to aXe

aXe's primary aim is to provide an easy introduction to text editing, but it is hoped that it is also sufficiently flexible to continue to meet most users' needs as their editing skills develop.

aXe can be used to edit several files at once, with each file being allocated to an editing buffer. An aXe window may support multiple buffers, but only one will be visible at any given time. The user determines the allocation of files to buffers, and hence to windows.

1.1 The components of an aXe window

An aXe window consists of five visible parts. From top to bottom they are:

`Filename` A line showing the name of the file currently being edited in the window. If a file has not been associated with this window then 'No name' will be displayed. The name is preceded by '*' if the text in the Editing buffer (below) contains unsaved changes. '~' at the beginning of the filename is shorthand for your home directory.

`Information bar`
A line that is updated with brief help explaining the action that would be performed if mouse button 1 were to be operated with the mouse pointer in the current position. If an action can be performed directly from the keyboard then the help line will contain a description of the keystroke(s) that would achieve it. It is enclosed in square brackets, e.g. [Ctrl-a]. See Section 1.5 [Using the keyboard for speed], page 3, for an explanation of what that means. The information bar is also used for displaying information arising from some actions, and for some error messages.

`Menu bar` A row of buttons having associated pulldown menus. Pressing and holding down mouse button 1 whilst a button is highlighted (by moving the mouse pointer onto it) will reveal the menu. The **Show** menu shows the names of all of the files being edited - the current file in the window from which the menu was popped up has a tick against it at the right hand side of the entry, and dividing lines are drawn to group the files associated with individual windows.

`Editing buffer`
An area displaying the text of the file that is currently being edited in the window. It may the topmost of a stack of buffers associated with the window, in which case a different buffer is made current by selecting its entry from the **Show** menu.

When an aXe window is resized all of the the adjustment in the vertical direction is applied to this component. Also, the size of the other components is discounted by the window manager and the information displayed during resizing is the size of the editing buffer in terms of characters, rather than pixels. In other words the editing buffer can only be resized in character increments.

`Minibuffer`
An area containing a menu button and a 1-line text buffer, called the minibuffer (See Section 2.5 [How to use the minibuffer], page 14.) The menu associated with the button provides convenient access to some of the more complicated facilities that are only available via the minibuffer. The selected ones feed the contents of the editing buffer to an external Unix command for processing and display the output, if any, in a fresh window.

The user can optionally suppress display of the Filename, Information bar and Minibuffer components. See Section 2.1.3 [suppressFilename], page 7, for details.

1.2 Basic editing

Traversing the text in the editing buffer and making changes to it can either be carried out from the keyboard or by using the mouse. The current position, or insertion point, in the text is denoted by a caret (`^`). It lies between characters and is initially right at the top of the file before the first character.

In order to edit using the keyboard, the mouse pointer - the symbol that moves around the screen as the mouse is moved - must lie somewhere within the editing buffer area. Note that the pointer symbol changes depending on which part of the screen it is over. When it is in an editable text area, i.e. the editing buffer or the minibuffer it will have the appearance of the letter `I`.

With the pointer in the editing buffer area, typing on the keyboard will cause the characters typed to be inserted at the insertion point. The character immediately to the left of the insertion point can be erased by using the `DELETE` key. The insertion point can be moved one line or one character at a time by using the cursor (arrow) keys.

An easier way to move the insertion point, especially if you want it to be some way away from the current position, is to move the mouse until the vertical bar of the pointer lies over the character which is to be to the right of the insertion point. With this done, quickly press and release the left hand button on the mouse (note that the information line is updated to show the line number of the chosen line). Slightly moving the mouse while the button is pressed down may result in one or more characters becoming highlighted. If this happens, you can either try again or simply ignore it for the time being.

1.3 Selecting text using the mouse

Some features involve the notion of a *selection*. A selection is identifiable as an area of highlighted text. You may already have accidentally created one by moving the mouse with the left button pressed down when attempting to adjust the insertion point. So, one way to create a selection is to move the mouse until the pointer is where you want the selection to start, then press and hold down the left mouse button. Now move the mouse with the button still held down and the text between the starting point and the current pointer position will be highlighted, i.e. selected. Release the mouse button when you are satisfied with the selection you have made.

If you didn't quite get it right, a selection can be adjusted at either end. Just move the pointer to the new end point and click (quickly press and release) the right hand mouse button, or press it down and sweep out the extension while continuing to hold the button down. Another way to make a selection is to move the pointer to where you want it to start and click the left hand mouse button, then move the pointer to the required end point and adjust with the right hand button. A selection can be cancelled by clicking the left hand mouse button within the buffer, but remember that doing so will likely affect the insertion point.

Once you have mastered using the mouse as just described you might like to try performing multiple clicks to make some selections. Rapidly clicking mouse button 1 the following number of times will make a selection as follows:

2. selects the word under the pointer. Words are delimited by Spaces, Tabs and Carriage Returns.
3. selects the line under the pointer.
4. selects the paragraph under the pointer. A paragraph boundary is defined by two Carriage Returns with only Spaces or Tabs between them.
5. selects the entire text.

The reason for making a selection is so that some operation can be applied to it, e.g. deletion, or copying to another part of the file. Copying can either be carried out by using the **Selection** entry from the **Insert** menu or by clicking the middle mouse button. The selection will be inserted at the current insertion point. The insertion point will be at one of the ends of the selection immediately after it has been made, so will normally have to be moved. Using the mouse to move the insertion point will cause the selection to be unlighted, but it will remain available for insertion until another one is made.

You should be aware that the selection mechanism is not peculiar to this application - many other X applications support it - but there can only be one selection across all applications at any one time. Selecting **Selection** from the **Insert** menu [the notation **Insert:Selection** for indicating a choice from a menu will be used in future], however, will result in the selection, whichever application owns it, being inserted at the current insertion point. And a selection made in this application can be transferred to some other by using its insertion method - probably clicking the middle mouse button.

1.4 Buttons instead of menus

After a time you may find it too slow to execute operations that you perform a lot using the menus. If this is the case then you might like to try the optional button interface in which an array of buttons - there is a button for every menu bar menu entry - is permanently displayed underneath the menu bar, making access to the operations that much faster. One way of requesting the button interface is to start aXe by using the command '`axe -buttons`'. See Section 2.1.3 [buttons], page 7, for an alternative.

To use a button simply click mouse button 1 when the button has been highlighted by moving the mouse pointer on top of it. It is not necessary to display the complete set of buttons - they do take up a lot of space. See Section 2.1.3 [buttonList], page 7, for details. A good compromise is to have your most frequently used operations available via buttons; the less frequently used will still be available via the menus.

1.5 Using the keyboard for speed

As you become even more proficient at editing you may want to speed things up by doing more from the keyboard. Many of the operations implemented via the menus (and buttons) have keyboard equivalents, e.g.

Ctrl-a Beginning Of Line

Ctrl-a means:- press and hold down the `CONTROL` key, and while it is held down press and release the `A` key, after which the `CONTROL` key can be released.

Other operations involve the use of the `META` key. However, keyboards don't have a key labelled `META`, so some other key, which varies from machine to machine, performs the role. It is `ALT` on

Acorns, `COMPOSE CHARACTER` on Vaxstations, and either of the ones at the ends of the Space Bar on Suns, viz. `LEFT` and `RIGHT` on Sun 3s, and the keys with a diamond shape as label on Type 4 and Type 5 keyboards. The `META` key is used similarly to the `CONTROL` key. See Section 1.5.1 [The default key bindings], page 4, for a list of the default bindings.

Some of the operations that are available via the menus and buttons do not have default bindings. See Section 2.3 [Actions specific to aXe], page 11, for a description of how they too can be associated with key strokes. The information there assumes a knowledge of how to use and specify X resources.

1.5.1 The default key bindings

The default key bindings are patterned after those in the EMACS text editor:

```
Ctrl-a  Beginning Of Line               Meta-b  Backward Word
Ctrl-b  Backward Character              Meta-f  Forward Word
Ctrl-d  Delete Next Character           Meta-i  InsertFile
Ctrl-e  End Of Line                     Meta-k  Kill To End Of Paragraph
Ctrl-f  Forward Character               Meta-q  Form Paragraph
Ctrl-g  Multiply Reset                  Meta-v  Previous Page
Ctrl-h  Delete Previous Character       Meta-y  Insert Current Selection
Ctrl-j  Newline And Indent              Meta-z  Scroll One Line Down
Ctrl-k  Kill To End Of Line             Meta-d  Delete Next Word
Ctrl-l  Redraw Display                  Meta-D  Kill Word
Ctrl-m  Newline                         Meta-h  Delete Previous Word
Ctrl-n  Next Line                       Meta-H  Backward Kill Word
Ctrl-o  Newline And Backup              Meta-<  Beginning Of File
Ctrl-p  Previous Line                   Meta->  End Of File
Ctrl-r  Search/Replace Backward         Meta-]  Forward Paragraph
Ctrl-s  Search/Replace Forward          Meta-[  Backward Paragraph
Ctrl-t  Transpose Characters
Ctrl-u  Multiply by 4       Meta-Delete             Delete Previous Word
Ctrl-v  Next Page           Meta-Shift Delete       Kill Previous Word
Ctrl-w  Kill Selection      Meta-Backspace          Delete Previous Word
Ctrl-y  Unkill              Meta-Shift Backspace    Kill Previous Word
Ctrl-z  Scroll One Line Up
```

In addition, the pointer may be used to cut and paste text:

```
Button 1 Down    Start Selection
Button 1 Motion  Adjust Selection
Button 1 Up      End Selection (cut)

Button 2 Down    Insert Current Selection (paste)

Button 3 Down    Extend Current Selection
Button 3 Motion  Adjust Selection
Button 3 Up      End Selection(cut)
```

Since all of these key and pointer bindings are set through the translations and resource manager, the user and the application programmer can modify them by changing the Text widget's translations resource (See Section 2.3 [Actions specific to aXe], page 11, for examples of the mechanism).

2 How to customise aXe

Like most X applications, aXe is highly customisable through resources. What follows makes the assumption that you know what resources are and how to apply the information that is presented. [N.B. I don't understand why extra space is generated around this introductory text.]

2.1 Resources specific to aXe

Before you can perform any customisation through resources it is necessary to know the widget structure of aXe. The hierarchy, including the first editing window, is shown below. All widgets are listed by instance and class name. Most of the widgets are custom widgets developed specifically for aXe. Where this is the case the superclass is enclosed in parentheses.

The file server window is either

```
fileServer  FileNominatorWindow (TransientShell)
```

or

```
server  TopLevelShell
```

with substructure

```
   table   XpTable
      waiter   Command
      show     AxeMenuButton (MenuButton)
      poleaxe  Command
      help     Command
```

The following menus are childern of the application shell, and are shared by all editing windows.

```
mMove   AxeSimpleMenu (Simplemenu)
mSearch AxeSimpleMenu (Simplemenu)
mInsert AxeSimpleMenu (Simplemenu)
mDelete AxeSimpleMenu (Simplemenu)
mHelp   AxeSimpleMenu (Simplemenu)
mMisc   AxeSimpleMenu (Simplemenu)
mWindow AxeSimpleMenu (Simplemenu)
mBuffer AxeSimpleMenu (Simplemenu)
mShow   AxeSimpleMenu (Simplemenu)
mFile   AxeSimpleMenu (Simplemenu)
mQuit   AxeSimpleMenu (Simplemenu)
mLogo   AxeSimpleMenu (Simplemenu)
mMini   AxeSimpleMenu (SimpleMenu)
```

Editing windows have the following structure apart from the instance name. Only the first, 00, is shown.

```
OO AxeWindow (TopLevelShell)
  panes AxeEditor (Composite)
      fileName   Label
      infoBar    Label
      buttons    XpTable
          moveMenu   AxeMenuButton (MenuButton)
          searchMenu AxeMenuButton (MenuButton)
          insertMenu AxeMenuButton (MenuButton)
          deleteMenu AxeMenuButton (MenuButton)
          helpMenu   AxeMenuButton (MenuButton)
          miscMenu   AxeMenuButton (MenuButton)
          fontMenu   AxeMenuButton (MenuButton)
               mFont   AxeSimpleMenu (SimpleMenu)
          windowMenu AxeMenuButton (MenuButton)
          bufferMenu AxeMenuButton (MenuButton)
          showMenu   AxeMenuButton (MenuButton)
          fileMenu   AxeMenuButton (MenuButton)
          quitMenu   AxeMenuButton (MenuButton)
          axeLogo    AxeMenuButton (MenuButton)
      edDeck     AxeTextDeck (Composite)
          ed         AxeText (AxeiiText) (AsciiText)
          ed         AxeText (AxeiiText) (AsciiText)
          ...
          ...
      miniMenu   AxeMenuButton (MenuButton)
      miniBuffer AsciiText
```

All of the above information, apart from the superclasses, can be obtained by running the editres application at Release 5 of X.

The menus consist of AxeSmeBSB (SmeBSB) objects. The names of the objects for each of the menus are these:-

```
mMove:   mbeg, mend, mup, mdown, mtop, mbot
mSearch: sbtext, sftext, sline, scaret, sbsel, sfsel
mInsert: ifile, isel, ipaste
mDelete: dword, dline, dsel, dcut
mHelp:   hgnrl, hbind, hcust, hnom
mMisc:   mundo, mwhere, mform, mcentre, mhcentr, mpref
mWindow: wnew, wfull, wclose, wclall, wicon, wdeicon
mBuffer: bempty, bfull, bclear, bclose
mFile:   fsvex, fsvall, fsave, fsvas, frvrt, fload
mQuit:   qquit, qsvex, qsvclw, qsvclb
```

Note that lists cannot be given for mFont and mShow, the showMenu buttons' menu. The composition of mFont is not known until startup time while that of mShow varies as windows are created and deleted. See Section 2.5.1 [The minibuffer menu], page 15, for a discussion of the minibuffer menu, mMini.

2.1.1 Application Resources

The following application resources are recognised

`fileServer`
: Setting this boolean resource causes aXe to start up in server mode with a FileNominator as the server window. See Section 2.8 [aXe as a server of windows], page 18, for more on aXe as a server.

`server`
: Setting this boolean resource, the default, causes aXe to start up in server mode. It has lower precedence than fileServer. If neither is set then the application terminates when the last window is closed. See Section 2.8 [aXe as a server of windows], page 18, for more on aXe as a server.

`windows`
: This resource is used to stipulate how many initial windows should be opened. If a greater number of files than the value of this resource is specified on the command line, then the value is ignored.

The associated command line keywords -noserver and -windows are supported. -noserver is equivalent to server:False; -windows requires a numeric argument.

The application geometry resource (-geometry keyword) is used as a default for all windows opened, but only the width and height parts, which are interpreted to refer to the size of editing buffers in characters, are used. It is also possible to apply geometry specifications one level down in the hierarchy at the individual window (AxeWindow) level. Windows are named 00, 01, 02, etc. If windows are deleted the lowest missing number is used as the instance name for the next window to be opened. This is only relevant if geometry resource specifications for named windows are in use, e.g.

```
Axe.windows:     2
Axe.00.geometry: 80x32+0+0
Axe.01.geometry: +0-0
```

2.1.2 AxeWindow widget resources

The following AxeWindow resources are recognised

`fileTitle`
: If this boolean resource is set to True then the name of a window will be set to the pathname of the file currently being edited in it, prefixed by '`axe:`'. The window's icon name is set to the last component of the path name, prefixed in the same fashion.

`fileTitlePrefix`
: This resource allows the user to alter the prefix that is used when the fileTitle resource is True.

`iconNamePrefix`
: This resource allows the user to alter the prefix that is applied to a window's icon name when the fileTitle resource is True.

2.1.3 AxeEditor widget resources

The following AxeEditor resources are recognised

`buttons`
: Setting this resource selects the default button interface, unless *buttonList* below is also set.

`buttonList`
: This resource enables the user to specify both which buttons are to be shown and what their layout is to be. Its value is a semi-colon separated list of rows of space separated button names. A '`-`' may be used in place of a button name in order to skip a column position. The names of the button widgets are presented below. *buttonList* implies *buttons*.

`focusToText`
: Setting this boolean resource has the effect of altering the keyboard focus so that keystrokes occurring in the name section, information bar or button area are redirected to the main text widget, i.e. the mouse pointer can be anywhere within the main application window for editing to take place.

`fontList`
: This resource defines the entries in a menu of fonts from which a choice may be made at run time to be applied to the visisble text buffer of the current window. It is a whitespace separated list of names. If a name contains a '`:`' then the left hand part will be used as the label of the entry and the right hand part will be taken as the name of a font, otherwise the whole name will be used for both purposes. The items in the list are in addition to the default entry **Default** which corresponds to the original font.

`infoTimeout`
: This resource specifies how long a message should remain in the information bar if no other message is generated in that time. The value should be given as a number of seconds. A value of 0, the default, disables the feature, thus each message remains displayed until the next is generated. The information bar reverts to displaying the version number if a timeout occurs.

`menuList`
: This resource is a space separated list of menu names specifying which menu buttons should appear in the menu bar and in what order. The default value is

  move search insert delete help misc font window buffer show file quit logo

  If logo doesn't appear in the list then it will be added at the end. Wherever it appears it will stretch to occupy any otherwise unused space.

`<menu>Menu`
: Resources for each of the move, search, insert, delete, help, misc, window, buffer, file and quit menus are defined so that the default contents may be changed. Their values are space separated lists of items consisting of object names from the entire pool of object names across the set of menus, plus '`+`' and '`|`'. '`+`' is interpreted to mean that the default menu should be substituted at that point in the menu, and '`|`' means that a dividing line should be added. See Section 2.1 [Resources specific to aXe], page 5, for the complete list of object names.

`miniMenu`
: This resource defines the entries to be displayed in the menu associated with the minibuffer. It is a whitespace separated list of names. If a name contains a '`:`' then the left hand part will be used as the label of the entry and the right hand part will be taken as the command to be run, otherwise the whole name will be used for both purposes. A '`+`' causes expansion of the default menu and '`|`' results in the generation of a dividing line in the menu.

`suppressFilename`
: This boolean resource is used to suppress display of the filename pane. This is most likely to be desired when the *fileTitle* resource (See Section 2.1.2 [AxeWindow widget

resources], page 7.) has also been set, otherwise the filename will be displayed twice in adjacent sections of the window. If both of these resources are set then the modification state of the file is also displayed in the title bar.

`suppressInfobar`
: This boolean resource is used to suppress display of the information bar pane. Of course, then, any error messages which would normally have been displayed there will be lost.

`suppressMinibuffer`
: This boolean resource is used to suppress display of the minibuffer.

2.1.4 AxeText widget resources

The following AxeText resources are recognised

`backupNamePrefix`
`backupNameSuffix`
: These resources only apply when *enableBackups*, below, is true. The backup name is the concatenation of the first, the filename, and the second, in that order. The defaults are the empty string and `.BAK` respectively.

`enableBackups`
: This boolean resource controls whether backups of files are made before they are saved for the first time.

`grabOnPopup`
: This boolean resource controls whether popups grab events. The default is False. When events are grabbed it is not possible to do anything elsewhere in the application until the popup is popped down.

2.1.5 AxeiiText widget resources

The following AxeiiText resources are recognised

`blinkPeriod`
: This resource specifies how long the matching '`(`', '`{`' or '`[`' is highlit when a '`)`', '`}`' or '`]`' is typed, and is expressed in milliseconds. The default is 500, i.e. half a second. Setting the value to 0 switches off the feature.

`blockCaret`
: This boolean resource specifies whether the default caret indicating the insertion point should be replaced by an inbuilt block caret.

WARNING: Since the introduction of *blockCaret* (above) and *caretBitmap* (below) it has come to light that the Tab key has no visible effect when either has been selected, although the Tab character is inserted into the file. Neither the cause nor a fix for this problem has been found.

`caretBitmap`
: This resource names the pathname of a file containing a bitmap that should be used as the caret that indicates the insertion point. This resource takes precedence over *blockCaret* if both are set.

`columns` This resource specifies the desired width in characters. It overrides any setting of the width in pixels.

`deleteOnInsert`
This boolean resource controls whether the entry of the first character via the keyboard (i.e. via the insert-char action) following the setting of the selection should delete the selection. The default value is False. This feature cannot be supported on older (than Release 5) versions of X, and setting *deleteOnInsert* will have no effect.

`expandTabs`
This resource specifies whether tabs should be expanded into spaces or not. The default is False. The resource only has an effect if the Tab key is bound to the action *insert-or-expand-tab* - which it is by default. See Section 2.3 [Actions specific to aXe], page 11.

`rows` This resource specifies the desired height in characters. It overrides any setting of the height in pixels.

`tabEvery` This resource is used to request that tab stops be set every *tabEvery* character positions. The default value is 8.

`undo` This resource specifies whether undo information should be kept. The default is True. See Section 2.7 [How the undo mechanism works], page 17.

`undoLevel`
This resource specifies for how many recent operations undo information should be kept. The default value is 1. A value of 0 implies keeping unlimited information, but remember that doing this could incur excessive memory usage costs.

2.2 Examples of resource settings

Here is an example in which the **New + Load** entry of the **Window** menu is replicated on the **File** menu:

```
*fileMenu: + wfull
```

and here is one in which the file menu items are rearranged:

```
*fileMenu: fload frvrt | fsvex fsvall fsave fsvas
```

The buttons in the button interface are AxeCommand widgets. Their names are identical to the corresponding menu entry AxeSmeBSB object names. Thus, the following resources would be sufficient to change the labels of the two **Load** entries to **Edit**.

```
*fload.label: Edit
```

Prefixing that resource specification by *XpTable or by *buttons would be sufficient to refer to just the button widget named **fload**.

Here is an example of how to specify the **buttonList** resource.

```
*buttonList: sbsel sfsel    -   dsel   -   mform wnew   - - fsave; \
                  -  scaret  isel dline  -      -    wclose
```

The ';' after fsave terminates the first row of buttons and the '-'s indicate empty column positions, so the layout of the resource value attempts to mimic the actual layout as it will appear in aXe's windows.

2.3 Actions specific to aXe

The main editing subwindow of a top-level aXe window is derived from an Athena Text Widget. Hence all of the default bindings of keys to Text Widget action routines apply. The Text Widget is fully described in 'Athena Widget Set - C Language InterFace' of the 'MIT X Window System Manual Set', but for your convenience an extract from the section describing the default key bindings can be viewed on-line (See Section 1.5.1 [The default key bindings], page 4.). aXe extends the functionality of the Text Widget by providing additional action routines, but it does not set up any default key bindings for them, i.e. the new features are only available through the menu and button interfaces, or the minibuffer. If you wish to set up your own bindings then you need to know the names of the new action routines.

They are

```
goto-line
search-caret
forward-search-selection
backward-search-selection

include-file
include-selection
paste

delete-word
delete-line
cut

undo
where
centre-line
set-preferences

new-window
close-window
close-all
iconify-all
deiconify-all

new-buffer
clear-buffer
close-buffer

save-and-close
save-all
save-file
save-as
load-file
reload-file
```

```
size
match-parens
find-match
re-search
forward-line
backward-line
delete-selection-or-previous-character
insert-or-expand-tab
keymap

set-mark
highlight-region
kill-region
copy-region
yank-region
```

It should be obvious which menu entries/buttons those in the first eight groups correspond to.

include-file inserts a file using the file selector, as distinct from insert-file, the Text Widget action bound to Meta-i that brings up the Text Widget's own insertFile dialogue box.

new-window and new-buffer may be given the optional argument 'load', causing the file selector to be popped up for the new window or buffer immediately it is created.

Without an argument save-and-close saves and closes all buffers. With the argument 'window' it saves and closes the current window; with the argument 'buffer' it saves and closes the current buffer.

The remaining actions are not available via the menus or buttons and hence must be bound to keystrokes or typed into the minibuffer (See Section 2.5 [How to use the minibuffer], page 14.).

size reports the size of the editing buffer in the information bar.

match-parens, if bound to '`)`', '`}`' or '`]`', causes the matching closing bracket to be highlit for a period of time determined by the value of the *blinkPeriod* resource (See Section 2.1.5 [AxeiiText widget resources], page 9.).

find-match causes a search to be made for the character matching the '`(`', '`)`', '`{`', '`}`', '`[`' or '`]`' immediately to the right of the caret, and if successful, highlighting the text between the two characters, inclusive.

re-search is like the standard search action except that regular expression searching is performed. It must be supplied with a direction argument of 'forward' or 'backward'. search and re-search share the same popup. The main observable difference when regular expression searching is in effect is that its title is 'RE search' rather than 'search'. There are also minor differences in the messages

The forward/backward-line actions are similar to the Text Widgets's next/previous-line actions except that they maintain position within the line if at all possible, i.e. the cursor does not gravitate towards the left margin as ever shorter lines are encountered.

The delete-selection-or-previous-character action does what its name suggests.

insert-or-expand-tab is bound to the `TAB` key by default, and operates in conjunction with the *expandTabs* resource (See Section 2.1.5 [AxeiiText widget resources], page 9.). If that resource is False then tabs are inserted, otherwise an appropriate number of spaces to take the insertion point up to the next tab stop is inserted.

The keymap action enables an alternative set of bindings to be switched in. keymap takes a name as argument, which, together with the suffix Keymap, defines the name of a translations resource. A name of None restores the original translations, e.g.

```
*ed.translations:#override \n <Key>F1:keymap(foo) \n ...

*ed.fooKeymap.translations:<Key>F1:keymap(None) \n ...
```

The final set of actions allows selections to be manipulated via the keyboard. set-mark sets an invisible marker. It is initially at the beginning of the buffer. highlight-region highlights the region between the marker and the caret. kill-region deletes the region between the mark and the caret, and stores it in cut buffer 2. copy-region copies the region between the mark and the caret to cut buffer 2. yank-region inserts the contents of cut buffer 2 at the current insertion point.

2.4 Examples of binding actions to keystrokes

Binding actions to keystrokes involves setting the translations resource of the Text Widget (instance name ed). Here is an example of how this might be done in a resources file.

```
Axe*ed.translations:   #override \n\
                       <Key>Down:   forward-line()  \n\
                       <Key>Up:     backward-line() \n\
                       <Key>Delete: delete-selection-or-previous-character()
```

Some of the features in aXe are implemented through setting ed's translations resource in the class resource file, but at Release 4 of X if you provide a value for the same resource it will completely override the application's. A mechanism, which aXe uses, has become available at Release 5 to overcome this difficulty. So if your aXe is an R5 one then you can skip the remainder of this section.

If at R4 you don't want to lose any of the application's bindings then you will have to repeat in your value those you wish to keep. The features affected are the matching of parentheses and the reporting of the current line number on a button release. This is how they are implemented in the class resource file

```
Axe*ed.translations:   #override \n\
                       <Key>): match-parens() \n\
                       <Key>]: match-parens() \n\
                       <Btn1Up>: extend-end(PRIMARY, CUT_BUFFER0) where() \n\
                       <Btn2Up>: where() \n\
                       <Btn3Up>: extend-end(PRIMARY, CUT_BUFFER0) where()
```

and the following is an example of how a user supplied value might look

```
Axe*ed.translations:   #override \n\
                       <Key>): match-parens() \n\
                       <Key>]: match-parens() \n\
                       <Key>Down: forward-line() \n\
                       <Key>Up:   backward-line() \n\
                       <Key>Delete: delete-selection-or-previous-character()
                       Meta<Key>=: where()
```

The matching of parentheses has been kept but the reporting of the current line number has been implemented another way, viz. by binding to Meta-=. The Btn?Up actions are not repeated because what is left after removing the where() action is the default behaviour that is defined by the Text Widget.

2.5 How to use the minibuffer

Commands corresponding to the two **Search:Text** operations plus **Search:Line**, **Insert:File**, **File:SaveAs** and **File:Load** can be entered directly into the minibuffer. The command names are styled after those of the Unix editor ed and are

```
/regular-expression
?regular-expression
[0-9][0-9]*
r filename
w filename
e filename
```

In addition pipe and shell commands have been introduced. Their names are '`|`' and '`!`' respectively.

Execution of pipe has the effect of replacing the selected text, if any, in the editing buffer by the result of filtering by the command specified in the remainder of the minibuffer. If no text is selected then the output of running the command specified as the pipe argument is inserted at the current insertion point.

Execution of shell has the effect of passing the whole buffer or the selected text to the shell command specified as argument and displaying the output, if any, of that command in a new aXe window, unless a '`>`' occurs in the command, in which case it is assumed that the user wishes to handle the output themself.

The command names must occur in column 1 of the minibuffer, and whitespace preceding the argument of '`r`', '`w`', '`e`', '`|`' and '`!`' commands is mandatory.

The minibuffer also recognises actions as commands. Thus it is not necessary restart aXe with new key bindings to experiment with untried features or to execute infrequently used actions - simply type them into the minibuffer, e.g. new-window() [or simply new-window] or new-window(load).

New action routines have been defined so that the minibuffer can be enabled from the editing buffer. When they are used the normal editing buffer translations are suspended and replaced by a minimal set of editing bindings that take effect in the minibuffer, i.e it is not necessary to move the mouse into the minibuffer to execute commands in it. If they are found to be inadequate the mouse can always be moved into the minibuffer to gain access to full range of Text Widget key bindings. The names of the action routines are

```
mini-search
mini-insert
mini-saveas
mini-load
mini-pipe
mini-shell
mini-stuff
```

mini-search may be given an argument of 'forward' or 'backward'; the default is forward. When the command is mini-insert, mini-saveas or mini-load the minibuffer is initialised with the directory already associated with the editing buffer. mini-pipe, mini-shell and mini-stuff may be given arguments with which to initialise the minibuffer, so frequently used filters and other minibuffer commands can be set up via user defined key bindings. If the last mini-pipe, mini-shell or mini-stuff argument ends with '`\n`' then the command will be executed right away.

Four other action routines have been defined. They are

```
mini-clear
mini-select
mini-commit
mini-abort
```

mini-clear enables and flushes the minibuffer. There is no command corresponding to **Search:Line** so mini-clear could be bound to a key for the purpose as it would provide an empty minibuffer awaiting the entry of a line number. mini-select enables the minibuffer without altering it, so the previous command can then be repeated by simply hitting the `RETURN` key. mini-commit executes the command in the minibuffer. mini-abort is defined so that a minibuffer command can be aborted while the accelerators are in effect. The default minibuffer translations and accelerators (the bindings that get installed as the editing buffer translations when the minibuffer is enabled via the above action routines) have `RETURN` bound to mini-commit and mini-abort bound to both `ESCAPE` and `Ctrl-g`.

In all cases except mini-search, hitting the `RETURN` key causes reinstatement of the editing buffer translations after execution of the minibuffer command. The reason for this is so that repeated searches can be made simply by continuing to hit the `RETURN` key. Thus it is necessary to explicitly abort search operations.

Note that the caret is only displayed in the minibuffer when it is active, either because the mouse pointer has been moved into it or because it has been enabled by execution of one of the mini- action routines, and further that the caret in the editing buffer is disabled in the latter case, although it won't always be easy to spot that that has happened.

Here are some typical bindings that might be used

```
Meta<Key>G: mini-clear()
Ctrl<Key>S: mini-search()
Ctrl<Key>R: mini-search(backward)
Ctrl<Key>X,<Key>B: mini-select()
Ctrl<Key>X,<Key>O: mini-select()
Ctrl<key>X,Ctrl<Key>F: mini-load()
Ctrl<Key>X,Ctrl<Key>I: mini-insert()
Ctrl<Key>X,Ctrl<Key>W: mini-saveas()
Ctrl<Key>X,Ctrl<Key>X: mini-pipe("sed 's/^/> /'\n")
<Key>Print: mini-shell(lpr -Pprinter\n)
```

The second last example will cause each line of the selected text to be prefixed by '`> `', i.e. it might be used to annotate a section of a mail message or news article being replied to. It is best to quote the whole argument to mini-pipe because of the way it would otherwise be split on spaces and commas. aXe does try to piece the line together again but can't know whether a split was as a result of a space or a comma; it always assumes that it was a space.

The last example causes the selected text or the whole of the text in the editing buffer to be queued for printing by printer 'printer'. The minibuffer menu provides a few ready-made examples that are likely to be commonly used.

2.5.1 The minibuffer menu

The minibuffer menu provides easy access to a few selected external commands that it is hoped will be commonly required, e.g. printing, spelling checking, formatting and mailing. The mini-shell commands that implement them are:

```
mini-shell(lpr)
mini-shell(spell)
mini-shell(nroff)
mini-shell(mail)
```

Those commands only place the commands '! lpr', '! spell' etc. into the minibuffer (with its accelerators enabled) so it is possible to add options to them before proceeding.

For convenience, the menu also provides exec and abort entries; the first is simply equivalent to hitting `RETURN`, and causes the command in the minibuffer to be executed, while the second removes the accelerators so that keystrokes in the editing buffer will revert to being directed to that buffer instead of to the minibuffer.

A further three entries are identical to the first three except that the commands are entered and executed in one operation. They are identifiable by having `<RET>` in the entry label, which is meant to convey that the command contains the `RETURN` to cause it to be executed. It does not make sense to replicate 'mail' in this way since a recipient will always have to be added to the minibuffer command before execution.

Although probably useful as it stands, the minbuffer menu is merely a demonstration of what can be done. It is totally customisable by the user by defining the *miniMenu* resource. Its value is a whitespace separated list of items. If a '`:`' occurs in an item then the left hand part will be taken as the corresponding menu entry label and the right hand part as the minibuffer action, otherwise the whole item will be used for both purposes. If the action part begins with 'mini-' then it is left untouched, otherwise it is taken to be an argument to mini-shell. Furthermore, an item of '`|`' is interpreted to mean that a dividing line should be added to the menu at that position. An item of '`+`' means that the default menu should be substituted at that point, i.e. the user can either replace or add to the default menu. So

```
*miniMenu:who | +
```

would add an item called who, that would execute the who command, to the top of the default menu, and separated from it by a dividing line, whereas

```
*miniMenu:who
```

would generate a menu consisting of a single 'who' entry.

The resource value defining the default menu is:

```
print:lpr spell nroff mail | \
exec:mini-commit abort:mini-abort | \
print<RET>:lpr\n spell<RET>:spell\n nroff<RET>:nroff\n
```

Labels and actions can be quoted using either the single or the double quote characters, e.g

```
"laser printer":'lpr -Plaser'    or    'laser printer':"lpr -Plaser"
```

The " and ' characters themselves can only be included by using the other quote character to do the quoting, e.g.

```
"jim's printer"
```

You can provide help to be displayed in the information bar when the mouse enters your menu items by setting each's help resource. The name of the widget is the same as the item's label, e.g.

```
*laser printer.help:Send to laser printer
```

2.6 Defining and using a keyboard macro

A single global keyboard macro can be defined. The following new action routines have been defined in this connection:

```
start-macro
end-macro
exec-macro
repeat-macro
```

start-macro and end-macro should be self-explanatory. exec-macro executes the macro once. repeat-macro optionally takes an argument specifying a repeat count. This is most likely to be of use when the action is executed as a minibuffer command. Otherwise a repeat count is taken from the minibuffer and the macro is executed that many times. This is most likely to be of use when repeat-macro() is used from a key binding. If the conversion fails or the minibuffer has been suppressed then the repeat count defaults to 1.

Here are some bindings that might be used:

```
Ctrl<Key>X,<Key>(: start-macro()
Ctrl<Key>X,<Key>): end-macro()
Ctrl<Key>X,<Key>E: exec-macro()
Ctrl<Key>X,<Key>R: repeat-macro()
```

A macro is defined by executing start-macro and stepping through the sequence of actions that will constitute one invocation, followed by end-macro. The macro is effective in all buffers, not just the one in which it is defined, and gets replaced if start-macro is executed again. With these bindings the effect of the mini-pipe example above could equally well be accomplished, after moving the insertion point to the beginning of the first line to be annotated, by defining a macro with the following sequence of keystrokes

```
C-x (
>
<space>
C-a
C-n
C-x )
```

and by executing `C-x e` for each subsequent line to be annotated, or by entering the number of remaining lines to be annotated in the minibuffer and executing `C-x r`.

2.7 How the undo mechanism works

The availability of undo is controlled by the resource *undo*, which is True by default. By default you are able to undo only the most recent operation that changed the file. If you wish to retain a greater history then set the *undoLevel* resource (See Section 2.1.5 [AxeiiText widget resources], page 9.) to the level you require. The default is 1. A value of 0 means keeping unlimited information, but that could run away with large amounts of memory if editing over an extended period. The new action undo performs a single undo.

Inserted characters, apart from control codes, which are dealt with individually, are handed back in units of uninterrupted runs of at most a line. A run is interrupted by any operation other than the insertion of a character, even if it doesn't change the file, e.g. moving the caret. If you want to undo single characters then delete them in the normal way. Deleted characters can be undone individually.

The implementation allows undos of undos. Once *undoLevel* consecutive undos have been performed the next (consecutive) undo will cause the most recent undo to be undone, the next the undo before that, and so on ad infinitum until some action other than undo breaks the sequence. Note that if you have a number of undos to perform it is quicker to type undo in the minibuffer and repeatedly re-execute it by hitting the `RETURN` key than by selecting **Undo** from the **Misc** menu each time.

You should be aware that currently multiple changes arising from performing a single action, e.g. by doing a 'replace All' or executing a keyboard macro, require multiple undos to get back to the position prior to the action being invoked, and may be impossible if *undoLevel* is not great enough.

If you find that for some reason the undo information has got out of step then the easiest way to get back in sync is to save and reload the file. Loading or reloading a file causes all the current undo information to be thrown away and the mechanism to be reinitialised.

2.8 aXe as a server of windows

There are two main differences when aXe is acting as a server: an additional window is created and aXe does not terminate when the number of windows reduces to zero. The function of the extra window is to provide a way of getting an editing window when there are no others to use as a launch pad, and to provide a way of shutting down the application. The default server window allows empty windows to be created and provides access to the help system. Alternatively, a FileNominator (See Section 3.1 [The file selector], page 19.) can be used as the server window by setting the top level *fileServer* resource (See Section 2.1.1 [Application resources], page 6,).

Additionally, editing windows can be obtained from a server by cooperating clients on the same host. Two such clients are provided: coaxe and faxe. Coaxe waits until it gets notification that its aXe window has been closed, whereas faxe terminates as soon as it has successfully contacted its server. Faxe can take several filename arguments, but coaxe is restricted to one. By default multiple files passed to faxe are assigned a window each. A group of files can be assigned to buffers of the same window by preceding the list by any leading substring of the keyword -buffer. The keyword may occur more than once on the command line. Coaxe and faxe start up aXe in non-server mode if a server is not already running, and coaxe is notified when the invocation terminates, i.e. when the last window is closed.

A display can support servers for many hosts, but only one server per host is catered for. If a second server is started for some host when one is already running then the second will take over the role of server to the client programs, and the first will be uncontactable by them.

Shutting down aXe from the server window is equivalent to doing a **Window:Close All**, i.e. a check is made for unsaved changes if any editing windows exist. A third client program, poleaxe, is also provided for shutting down the server, but checks are not carried out when it is used. It is not a serious client; it only came into being because of the pun afforded by its name.

3 aXe's secondary popup windows

From time to time as aXe is running windows other than the main editing windows and the initial server window, if running in server mode, that is, are popped up for the input and output of information. Each type is briefly described in one of the sections that now follow.

3.1 The file selector

The FileNominator popup is popped up whenever a menu operation that requires a file name is selected, or an equivalent action routine has been activated through a key binding set up by the user.

The FileNominator is both a directory browser and a file selector. It consists of a scrollable list of the contents of the directory associated with the buffer from which it was popped up, or the directory last visited by the user in earlier usage, a filter section, which allows the user to filter the list of files displayed according to normal shell pattern matching rules, a 1-line text area in which the filename to be returned is assembled, and three buttons. The filter pattern that is initially applied may be something other than the default '*'. A different value is specified by setting the value of the resource *filter*, for example to '*[ch]'.

A file may be nominated by choosing from the list using mouse button 1 (causing the name to be entered into the filename input area), or by typing into the input area, or by doing both. All keyboard input is directed there when the list widget has the input focus. The content of the input is assumed to be relative to the directory currently displayed if not an absolute pathname. The selection is always taken from the nomination in the input area, and happens when the select button is clicked (mouse button 1) or the `RETURN` key is pressed. If no modification is to be made then a choice from the List can be selected by double clicking mouse button 1 or by single clicking mouse button 2. Further, an immediate selection of some other directory in the path of the displayed directory can be made by selecting from the 'path' button menu.

A '~' as the first character in the input area is understood to mean the user's home directory. If a nomination begins with a '`$`' then the first component will be treated as a possible reference to an environment variable, with the expansion being made if a variable of that name exists.

In addition the user may define a list of potential nominations - either files or directories - to be added to the 'path' button menu. Since selecting an entry from the menu is equivalent to entering the entry in the input area and selecting it in the normal way, i.e it is subject to the checks that would be applied if the user had typed the value as a nomination, then '~' and '`$`' may feature in the user-added entries. The user adds entries by providing a value for the *pathList* resource of the FileNominator. It is a colon separated list of pathnames that are converted into menu entries and added to the path button menu.

When the selection is a directory the contents of the directory are filtered according to the pattern in the filter input area, so that a reduced subset of interest may be displayed. A new filter can be run on the current directory by hitting `RETURN` in the filter input area or by selecting Apply from the menu belonging to the filter menu button. The menu has two other entries: 'Apply to directories', which controls whether directories should always be displayed or whether they should also be subject to filtering, and 'Apply to `"."` files', which controls whether names beginning with a '`.`' should be displayed or not. The last two entries are toggles, whose selection cause toggling of the setting and rescanning of the directory with the toggled value effective. The `"."` files entry takes precedence over the one for directories e.g. if the pattern is '*c' and there is a file called .filec, then it won't be shown unless there is a tick mark against the 'Apply to "." files' entry.

3.2 The numeric keypad

The NumericPad popup is popped up whenever the **Line** entry of the **Search** menu is selected, or the goto-line action invoked by other means. This popup is an instance of a general purpose widget for entering numeric data, and contains features that are probably inappropriate for aXe, like the ability to change the base to octal or hexadecimal.

A number can be entered into the 1-line text area by clicking the digit buttons or from the keyboard whilst the popup has the input focus. A number is submitted to the application by clicking the enter button or by hitting either the `RETURN` or the `ENTER` key. The cancel button is for dismissing the popup without entering a number.

3.3 The control code selector

The ControlCodeSelector popup is popped up whenever the **Control Codes** entry of the **Insert** menu is selected, or the insert-control action is invoked by other means. Selecting an item in the table of control codes that is displayed causes the item to become highlighted and its ASCII code to be displayed in a label at the lower left of the popup. The widget that is at the heart of this popup has a resource, *base*, which controls how the numeric value of the code is displayed. aXe displays the value to the base 8, i.e. in octal.

The currently selected control code is entered into the current buffer of the window from which the popup was popped up by clicking the enter button with mouse button 1. Several consecutive control codes can be entered as the popup is only popped down when the cancel button is clicked. A code from the table can be entered directly into the current buffer by clicking the item with mouse button 2.

3.4 The confirmer

The Confirmer popup is popped up whenever the user is about to perform a potentially destructive act, like clearing a buffer that has unsaved changes. Apart from a superfluous bitmap, it contains a message describing the reason why the user is being asked to confirm proceeding with the requested operation, and three buttons. The rightmost button - cancel - is for dismissing the popup by abandoning the requested operation. The remaining two buttons provide alternative courses of action. If the middle button is greyed out then it means that there is only one possible action to take.

The symbols at the right hand end of the button labels signify keyboard accelerators for selecting those buttons. `^A` is shorthand for hitting key `A` in conjunction with the `CONTROL` key. Thus, if the popup is popped up as a result of a keyboard event then in all probability the user will be able to make a choice without having to resort to the mouse since the popup is centred around the mouse pointer and a popular way of working is for the widget having the input focus to be determined by the location of the mouse.

3.5 The preferences setter

The Preferences popup allows a selected number of user preferences (resources) to be changed at runtime. You should note that under 'Wrap Mode' selection of 'Auto Fill' automatically forces on 'No Wrap', and selection of 'Line' or 'Word' automatically forces off 'Auto Fill'. These are the combinations that it has been found work best.

Any typing within the popup is directed to the 'Tab Every' entry field as this is the only one which requires keyboard input.

Although there is apparently provision for having the settings apply to just one buffer, all the buffers in one window, or to all buffers, only the first is possible. The buttons corresponding to the last two cases are insensitive to button presses, which is indicated by their being greyed out.

3.6 The help system

The on-line help is presented by running a separate application whenever an entry is selected from the **Help** menu, or the Help button in the server window is clicked. This document and the on-line help contain basically the same information, being generated from the same source file with only a few minor changes here and there - for instance the information in this section is different in the on-line version.

The application, called axinfo, although principally written for use by aXe, isn't strictly part of aXe and so is not described here. It is a standalone program that could be used to display other documentation developed in the same way.

Very briefly, the way it works is as follows: the help is organised for hypertext lookup, i.e. words in the text are highlighted and made sensitive to mouse clicks, and performing that action displays some further text referred to by the highlighted text. In other words the user can jump around the help in an order that suits them.

Should there be any doubt about how to use the program its own hypertext help is accessible when being used to display aXe's, i.e. you find out how to use the program by using it.

If a window fails to appear in response to requesting help then it is almost certain that the executable, axinfo, was not found on your shell command search path. If a small window appears containing a message that the application defaults file could not be found then axinfo has either not been installed properly or you have incorrectly overridden the default search path for application defaults files, for example by setting the environment variable XFILESEARCHPATH,

Once axinfo has been started up it is independent of aXe, i.e. terminating the one does not terminate the other. On the other hand, if an axinfo is already running, as a result of having been started independently or remaining from an earlier invocation of aXe, when help is requested then aXe communicates with that axinfo rather than fire up a new one.

4 aXe's extension language

aXe's extension language is Tcl. As Tcl is a separate package with its own documentation, which should be available locally, it is not described here. The language extension is an option at build time. If the **Help** menu does not contain an **Extension** entry then the option has not been configured into the version of aXe you are running.

In order to integrate Tcl into aXe a new action routine, execute, has been introduced. It takes a Tcl command as argument. Thus Tcl commands can be bound to keystrokes, e.g.

```
*ed.translations:#override \n\
                 Ctrl<Key>x,<Key>r:execute(prefix)
```

where 'prefix' might be the following procedure:

```
proc prefix {{text {> }}} {
    with-selection action insert-string $text
}
```

with-selection is a general purpose procedure that applies another procedure to each line of the selection in turn. The applied procedure and its arguments are passed as arguments to with-selection. In the example above the applied procedure is 'action' and its arguments are 'insert-string' and '$text'. with-selection is implemented as follows:

```
proc with-selection {func {args {}}} {
    set selection [get-selection]
    set start [lindex $selection 0]
    set end   [lindex $selection 1]
    set lines [lindex $selection 2]

    if {$start == $end} {return}

    set-position $start
    for {set line 0} {$line < $lines} {incr line} {
        action beginning-of-line
        eval $func $args
        action next-line
    }
    action beginning-of-line
    set-selection $start [get-position]
}
```

By default prefix prepends '`> `' to each line in the selection, but by supplying an argument a different prefix can be employed, e.g.

```
prefix "! "
prefix "    "
```

prefix and with-selection make use of a number of builtin commands (See Section 4.1 [Builtin extension language commands], page 22.) that have been defined.

4.1 Builtin extension language commands

`action` *arg1 arg2 ...*
: enables action routines to be executed from within Tcl commands. Action routine arguments are separated by spaces, not enclosed in parentheses, e.g. search forward, not search(forward).

`eval-buffer`
: evaluates the current buffer on the assumption that it contains a Tcl script. In other words new commands can be entered and immediately tried out.

`get-selection`
: returns a list consisting of the starting and finishing positions of the selection, the number of lines spanned by it, and the selection itself.

`get-position`
: returns the value of the insertion point.

`set-position` *point*
: sets the insertion point to the value of its argument.

`set-selection` *start finish*
: sets the selection to be the text spanned by the positions passed as arguments.

`set-value` *resource value*
`set-source-value` *resource value*
`set-sink-value` *resource value*
: set the resource named in the first argument to the value in the second for the text widget or its source or sink objects.

4.2 Other extension language commands

A small collection of commands built using the builtins (See Section 4.1 [Builtin extension language commands], page 22.) and action routines should have been loaded when aXe was started up. It consists of the following commands.

`with-selection` *proc args*
: applies a procedure and its arguments to each line of the selection in turn.

Example:

```
with-selection action insert-string "> "
```

`prefix` *string*
: prepends an optional string to each line of the selection. The default string is '`> `'.

`strip` *n*
: removes a number of characters equal to the value of its argument from the beginning of each line in the selection. If no argument is given the default is 1.

`join-lines` *n separator*
: joins together the next n lines as given by the first argument (default 1) to the current line, and inserts the text given by the second argument (default `" "`) at the join.

`format-all`
: formats the entire buffer by applying the action form-paragraph to successive paragraphs. On completion the caret is restored to the character position it was originally at, but of course that is likely to be correspond to a different place in the text.

`upper`
`lower`
: convert the selection to upper and lower case respectively.

4.3 Using the extension language

User-defined commands that are to be loaded at startup should be stored in the file axe.tcl in the user's home directory. Commands can also be loaded dynamically at runtime from a file by executing '`source pathname`' in the minibuffer. If the file does not exist then the command will fail, and that failure will be reported identically to other errors, i.e. the keyboard bell will be rung and an appropriate message displayed in an alert popup.

The easiest way to execute Tcl commands is to type them into the minibuffer. Thus the example given at the beginning (See Chapter 4 [aXe's extension language], page 22.) could equally well be executed by typing '`prefix`' in the minibuffer, whereas making a selection and typing

```
prefix "    "
```

will cause each line in the selection to be indented by 4 spaces.

Using the minibuffer is especially convenient if you want to create and try out new commands from within aXe. If the current buffer contains a Tcl script defining some new procedures then these can be loaded into aXe by typing '`eval-buffer`' in the minibuffer. The new procedures may be tried out by issuing commands in the minibuffer to invoke them. Changes to the script can be made and the process repeated as often as necessary as defining a procedure replaces the existing definition, if any.

The extension language makes it possible for things to be done in aXe that either I couldn't envisage users wanting to do or wouldn't want to implement as fixed features. Here are just a couple of examples that might set you thinking of the possibilities.

`Example 1`

The following procedure demonstrates how the extension language might be used to set up modes.

```
proc a-mode {} {
    set-value tabEvery 4
    set-value wrap word
}
```

Execution of the procedure will affect the current buffer of the window in which it is executed. There are three ways in which it could be executed:

1. by typing '`a-mode`' in the minibuffer.
2. by setting up a minibuffer menu entry and selecting it. The entry might be set up using the following resource specification:

   ```
   *miniMenu:+ 'A mode':mini-stuff(a-mode\n)
   ```

3. by setting up a translation, such as the following:

   ```
   *ed.translations: #override \n\
                     Meta<Key>m:execute(a-mode)
   ```

   and typing `Meta-m` in the editing buffer to which it is to be applied.

`Example 2`

This example shows how a procedure defining a C++ template header file might be set up.

```
proc class {class {baseclass {}}} {
    action insert-string "#ifndef ${class}_h\n"
    action insert-string "#define ${class}_h\n\n"
    action insert-string "class $class"
    if {$baseclass != {}} {
        action insert-string " : public $baseclass"
    }
    action insert-string "\n\{\npublic:\n"
    action insert-string " $class\(\);\n ~$class\(\);\n\n"
    action insert-string "protected:\n\n"
    action insert-string "private:\n\};\n\n#endif\n"
}
```

When invoked with two arguments, the name of the new class and the name of the base class, then the following is generated at the insertion point of the current buffer, assuming arguments of Foo and Bar:

```
#ifndef Foo_h
#define Foo_h

class Foo : public Bar
{
public:
    Foo();
    ~Foo();

protected:

private:
};

#endif
```

If the second argument is not given then the '`class`' line becomes

```
class Foo
```

Concept Index

Action Index

U

W

Y

Resource Index