[15] Adobe Illustrator. Tutorial. Adobe System Incorporated, 1992

References

[1] Brian W. Kernighan. PIC - A language for Typesetting Graphics. Software - Practice and Experience, vol. 12, pp. 1-21 (1982)

[2] Narain Gehani. Document Formatting and Typesetting on the UNIX System. Second edition. Silicon Press, 1987.

[3] Kenneth H. Rosen, Richard R. Rosinski, and James M. Farber. UNIX SYSTEM V Release 4. An introduction for new and experienced users. Osborne Mc Graw-Hill 1990.

[4] David Barron and Nike Rees. Text Processing and Typesetting with UNIX. Addison-Wesley 1987

[5] UNIX Programmer's Manual. Vol 1-2, Holt, Reinhart, and Winston, 1983

[6] Brian W. Kernighan and Rob Pike. The UNIX Programming Environment. Prentice Hall, 1984

[7] Marc J. Rochkind. Advanced UNIX Programming. Prentice Hall, Inc, 1985

[8] Adrian Nye. Xlib Programming Manual. O'Reilly & Associates, Inc, 1990

[9] Xlib Reference Manual. O'Reilly & Associates, 1991

[10] Adrian Nye and Tim O'Reilly. XToolkit Intrinsics, Programming Manual, OSF/Motif Edition. O'Reilly & Associates, 1990

[11] XToolkit Intrinsics, Reference Manual. O'Reilly & Associates, 1990

[12] Dan Heller. Motif Programming Manual for OSF/Motif Version 1.1. O'Reilly & Associates, 1991

[13] InterViews Reference Manual idraw, 1989

[14] User commands xfig, fig2dev

(c) the fill mode for a **box, circle,** or **ellipse**.

(d) the **arc** direction.

10. *o_size, o_nhead, o_symtab,* and *o_ddval* are not used by the WD-pic, but are needed for compatibility with the **Pic** compiler.

11. *o_fillval* is a fill value.

12. *o_val[]* is an array whose length is usually more than 1, and it contains additional information, depending on the element type:

    (a) for a **box**, the width and height.

    (b) for a **circle** or **ellipse**, the two radii, which are equal for a **circle**.

    (c) for a **line, spline,** or **arrow,** the number of parts, the relative coordinates of the end points, the absolute coordinates of the last point, and the width and the height of the arrowhead, if any.

    (d) for an **arc**, the coordinates of the end points, and the width and the height of arrowhead, if any, and the radius.

    (e) for a **text** the width and height.

B.3 String array

Each element of the string array contains following fields:

int t_type;
char t_op;
char t_size;
*char *t_val;*

here,

1. *t_type* is the string's adjustment, which may be LDJUST, RDJUST, ABOVE, BELOW, or CENTER.

2. *t_op* and *t_size* are not used, but are needed for compatibility with the **Pic** compiler.

3. *t_val* is a pointer to the string.

int o_nobj;
int o_mode;
float o_x;
float o_y;
int o_nt1;
int o_nt2;
int o_attr;
int o_size;
int o_nhead;
*void *o_symtab;*
float o_ddval;
float o_fillval;
ofloat o_val[1];

Here:

1. *PicTextUnion* is a pointer, which contains either the address of the element of the **Pic** text list in WD-pic itself or the index of an element in the **Pic** text array in the **Pic** compiler.

2. *o_type* is an object type. It may be BOX, LINE, ARROW, CIRCLE, ELLIPSE, LINE, SPLINE, MOVE, or TEXT.

3. *o_count* is the number of elements in the array *o_val*, which is described below.

4. *o_nobj* is the index of the element in the object array.

5. *o_mode* is the direction for drawing the element. It may be R_DIR, for the right direction, L_DIR, for the left direction, UP_DIR, for the up direction, and DOWN_DIR, for the down direction.

6. *o_x, o_y* are the coordinates for positioning the element; for a **box, circle,** or **ellipse** they are the coordinates of the geometric center, for a **array, line, spline, arc,** or **move**, they are the coordinates of the beginning, for a **text** they are coordinates of the left or right boundary or the center, depending on the **text**'s adjustment type.

7. *o_nt1* is the index of the string array element that contains the first text string that belongs to the object.

8. *o_nt2* is the index of the string array element that contains the first text string that does not belong to the object. If no text string belongs to the object, then *o_nt1* and *o_nt2* have the same number.

9. *o_attr* contains the following information :

    (a) the kind of arrowhead for a **line, spline, arrow,** or **arc**.

    (b) the line style.

Appendix B

B.1 Pic text list

Each element of the **Pic** text list contains the following fields:

*char *curString;*
int curStringLength;

where,

1. *curString* is a pointer to the string written in the **Pic** language.

2. *curStringLength* is the number of characters in this string.

Here are examples of strings:

- corresponding to a **Pic** element:

  "*A1: box dotted width 1*"

  or "*circle rad 1 "CIRCLE"* "

- corresponding to a **Pic** operator:

  "*for* $i = 1$ *to 10 by 2 do* {
  circle radius $0.5 * i$
  }"

- corresponding to a **Pic** default size change:

  "*boxwid = 1.5*"
  or "*reset boxwid*"

B.2 Object array

We can see below a full description of the object array element structure with the changes, necessary for using it in the WD-pic:

```
union {
    PicTextDef *Ptr;
    int Num;
}PicTextUnion;
int o_type;
int o_count;
```

1. *Loop*: If this option is selected, then a dialog pops up on the screen in the center of main window. This dialog allows inserting the loop index, the high and low boundaries, and the step. After that, the user can press the "OK" push button and the preferred text editor is opened the same way as in the *Text edit* item of the *Edit* menu. But this time, the **Pic** text contains a string "for ... = ... to ... by ..." according to the user's choice. The user fills in the missing body of the loop. After the user exits the text editor, the changed picture immediately appears in the drawing area.

2. *If ... then ... else*: After selecting this option, the preferred text editor is opened immediately. The **Pic** text will contain a string "if ... then ... else". The user fills in the missing expression and commands. After the user exits the text editor, the changed picture immediately appears on the drawing area.

A12 *Defaults* popup menu

The popup menu *Defaults* contains three items, *Scale, New defaults,* and *Reset.*

1. *Scale* allows the user to change the picture's scale. When the user selects this item a new dialog pops up on the screen in the center of the main window. The user can insert a new scale value.

2. *New defaults* allows the user to change size defaults. When the user selects this item, a dialog pops up on the screen. The dialog contains a table of all current size defaults. The user can change any of these.

3. *Reset* allows the user to reset all defaults to their beginning values. When the user selects this option, a dialog pops up on the screen. The dialog contains a table of all default sizes that were changed. There is a toggle button near each size. Pressing a particular toggle button means to reset the corresponding size to its old default. This dialog also contains the toggle button RESET ALL. By pushing this toggle button, the user resets all default sizes to their beginning values. This affects only the part of the picture described by the text after the current insertion point. All changes take effect on the picture displayed right after editing this dialog.

2. *Insert:* If the user has selected this option then each new element will be added before the selected element, until the *End Insert/Add* option is selected.

3. *Add:* If the user has selected this option, then each new element will be added after selected element, until the *End Insert/Add* option is selected.

4. *End Insert/Add:* The user selects this option to finish an addition or insertion of new elements into the picture.

5. *Modify attribute:* If the user has selected this option, a new dialog appears in the center of the main window. It contains text window with the **Pic** string corresponding to the selected element. In the button panel, in the element parameters section appears a push button set corresponding to the selected element. The user can change element attributes directly in the text window or with the help of the element parameters push buttons. If he/she prefers the push buttons, all changes immediately appear in the text window and, of course, in the drawing area.

6. *Delete element:* Delete the selected element.

7. *Text edit:* WD-pic allows two ways to edit a picture, by graphic editing and text editing. The previous six options are for graphic editing. This *Text edit* option allows editing the text of the **Pic** internal representation. If environment variable EDITOR is defined in the containing UNIX system, for example EDITOR = vi, then after selection of this option, the editor named by EDITOR is opened on the current intermediate representation. The user can change the **Pic** text as he/she wants. After user exits from EDITOR, the resulting changed picture immediately appears on the drawing area, if of course, the changed **Pic** text is a correct **Pic** program. Otherwise, if the changed **Pic** text contains errors, then a new window, named "Errors and warning messages" is opened. Also, a dialog is opened that informs the user that a new file ERROR.pic was created and this file contains the changed **Pic** text. The original internal representation is lost.

A.10 *View* popup menu

The popup menu *View* contains one item: *File View.*

1. *File View:* After user has selected this option, a standard prompt dialog immediately appears in the center of the main window. This dialog asks the user for the name of a file. The user must insert a file name and press the "OK" push button. If the push button was pressed without having given a file name, then the user is presumed to want to see the **Pic** text corresponding to the current picture. The user can see desired text in a newly created window "View". This new window contains a "OK" push button. After this button is pressed, the window is destroyed.

A.11 *Structures* popup menu

The *Structures* popup menu contains two options, *Loop* and *if ... then ... else.*

When the user click on the **fill** push button the corresponding dialog is appeared The dialog contains a slider, with the help of which the user can set a gray scale value. An example of the corresponding **Pic** internal representation is the string "box *fill 0.5*"

A.7 Changing an element's size according to the size of last element of the same kind

When the user want to change the size of the current element according to the size of last element of the same size he/she need to click on the push button **same**.
An example of the resulting **Pic** internal representation is:

" box wid x
box *same*"

A.8 *File* popup menu

The *File* popup menu has the following items: *New, Save Current File, Store as New File, Read File,* and *Quit.*

1. *New:* Start a new picture.
2. *Save Current File:* If the current picture has a name i.e., it was saved before or it was loaded by *Read File*, then the **Pic** internal representation is saved with the same name, otherwise the user is asked for a new name and the **Pic** internal representation of the picture is saved in the file with the new name.
3. *Store as New File:* The user is asked for a name, and the internal representation of the current picture is saved in a file with this name.
4. *Read File:* The user is asked for a name of a file. If a file with this name exists and the file doesn't contain any syntax error, it is loaded as a **Pic** file, and then the represented picture is displayed in the drawing area. Otherwise, if the **Pic** file contains errors, then a new window, named "Errors and warning messages" displaying the error messages is opened.
5. *Quit:* Exit WD-pic.

A.9 *Edit* popup menu

The *Edit* popup menu has the following items: *Select element, Insert, Add, End Insert/Add, Modify attribute, Delete*, and *Text edit.*

1. *Select element:* After choosing this option, the user can select any element on the drawing area. The selected element will be drawn by double line. After *Select element*, any of the *Insert, Add, Modify attribute,* and *Delete* options can be chosen.

- third option:

  " box
  ...
  circle rad z *at first box + (x, y)*"

For an **arrow, line, spline,** or **arc**, there is a **from-to** push button, which allows placing the beginning of the **arrow, line, spline** or **arc**, and its end. Selection of the push button invokes a dialog, that offers two instances of the element placement in order to

1. place the beginning of the object with the **from** option, and
2. place the end of the object with the **to** option.

After that, the same dialog as for a **box, circle,** and **ellipse** offers the user three ways to place beginning or end of the element, and the user must execute the same actions as in the previous case.
Here are a few examples of the **Pic** text generated for placement of the beginning and the end of a line:

- first option: "line *from (x,y)* up x1 left y1 *to (z,w)* ".
- second option:

  " *A1*: box
  ...
  A2: circle
  ...
  line *from A1.n* up x left y then down 2 *to B1*"

- third option:

  " box
  circle
  ...
  line *from first box to second circle.w*"

A.5 Changing Line Style

In oreder to change a line style of an element the user must click on the **lines** push button. The corresponding dialog offers four possible line styles: **solid**, **dotted**, **dashed** and **invisible**. For each option there is a toggle button. If no line style is selected, then the default **solid** is used.
For example, when the **dotted** line style is selected the substring *dotted* is added to the current place of the **Pic** internal representation.

A.6 Changing Fill Style

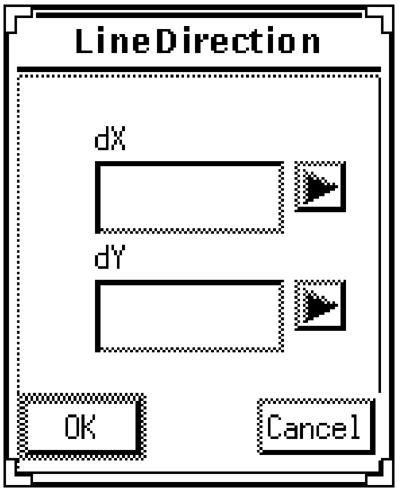

Figure 12: Direction dialog for **line, arrow** or **spline**

1. Place the element by its center, using the **at** option.
2. Place the element by one of its corners, using the **with** option.

There is a toggle button for each option. If the chosen option is to place the element by its corner, a new dialog is invoked, and it asks the user to point to the suitable corner with the left mouse button. If the chosen option is to place element by its center or if the user has chosen a suitable corner of the element, a new dialog appears. This dialog offers three ways to place the element,

1. by the coordinates of its center or corners.
2. relative to another, labeled element. If this element has no label yet, then label will be added automatically.
3. relative to a numbered object of chosen kind, for example relative to the first box.

After selection of the suitable option, the user must

- for the first option, choose one point in the drawing area
- for the second and the third options, choose an element and after that click on the required point. If this point is a corner of the object then in the **Pic** text, it will be entered as the element's corner.

Here are examples of **Pic** texts, generated by each option for circle placement.

- first option: "circle rad z *at x, y*".
- second option:
  " *A1*: box
  ...
  circle rad z *at A1 + (x, y)*"

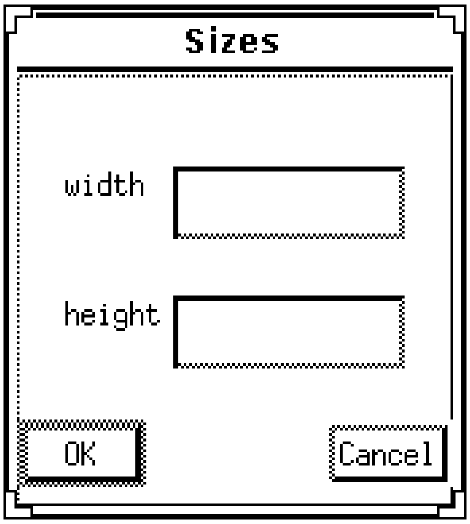

Figure 11: Sizes dialog for **box** or **ellipse**

substring *width x height y* to the string in the **Pic** text that describes the **box**.

The size of a**line, arrow,** or **spline** can be changed with the help of the **direction** push button. Selection of this push button calls a dialog, which is shown in Fig. 12.
The user can insert data directly into the text windows, labeled by “dX” and “dY”. Alternatively, the user can select an arrow button, corresponding to the current text window and choose one of the standard **Pic** sizes from a menu that pops up after the button is pressed. After pressing the “OK” push button, a grid is imposed on the picture with its cell sizes dX, dY, assuming valid dX and dY. The user can choose end points for all segments of a **line** or an **arrow** or end points for a **spline**'s guiding lines on this grid by the left mouse button. He/she can finish the task by pressing the middle mouse button or by choosing a new element from **Pic** tokens. After each chosen point, the **line, arrow**, or **spline** immediately changes its view. Simultaneously, the string like *left linewid * x up lineht * y* is added to the current place in the **Pic** internal representation.
For an **arc**, the **radius** push button invokes the same dialog as the **size** push button for a **circle** with the same action.

For a **move**, the **place** push button calls the same dialog as the **direction** push button for **line, arrow,** and **spline** with the same action, except that the user can choose only one point on the grid.

A.4 Changing the place of an element

In order to change a placement of a **box, circle,** and **ellipse**, the user need to click on the **place** push button. Selection of this push button invokes a dialog that offers two kinds of element placement.

Appendix A

User Manual

A.1 A new element addition

In order to add a new element to the picture user need to click on the corresponding push button from the **Pic** tokens part of the button panel. The new element is placed in the drawing area with default drawing and geometric parameters. This element is placed at the current insertion point. At any time, the accumulated drawing is centered in the drawing area. Simultaneously, a corresponding string is added to the **Pic** internal representation. If immediately after choosing some element, such as **box, circle** etc., the **text** push button is chosen, then **text** is inserted into the element. In order to create self-standing **text** user must press the ";" push button before pressing **text** push button in order that the current **text** not be considered as part of the previous element. For example, when the user clicks on the **box** push button, the string *box* is added to the current insertion point in the **Pic** file, the following picture is added to the current insertion point on the drawing area, and the whole accumulated picture is centered again.

A.2 Changing the direction of a picture

In order to change a direction of a picture the user must click on a arrow button whose arrow points to the desired direction. For example, when the user clicks on a arrow button whose arrow points to the right, the string *right* is added to the current place in the **Pic** file. The effect of this is to cause the picture to grow to the right from the current insertion point on the screen. If the current insertion point is not the end, then this may cause portion of the picture to change their orientation entirely when the picture is redrawn.

A.3 Changing the size of an element

In order to change a size of a **box, circle,** or **ellipse** the user need to click on the **size** push button. When this push button is pressed, a dialog appears in the lower left corner of the main window.

This dialog has two labeled text windows: for **box** and **ellipse**, they are **width** and **height**, for **circle**, they are **radius** and **diameter**. The dialog also has two push buttons "OK" and "Cancel". In Fig. 11, we see sizes dialog for a **box** or an **ellipse**. The user must insert the required parameters into the text windows. For a **circle**, he/she needs to insert only one of them. If the user defines the parameters and then presses the push button "OK" then the element in the drawing area immediately changes its size according to the inserted values. Simultaneously, the corresponding substring is added to the current place in the **Pic** internal representation. For example, result of changing a **box**'s size is the addition of the

The idea of laying out grids that have spacing equal to the value of pic variables was good, but is a pain to set up with the pop up window interface. It would be nice if the variables could be selected by mouse when they are simple, and have to be typed only when they involve expressions. Even better would be for the grid to have a default spacing of moveht vertically and movewid horizontally. Moreover, it would be nice if the origin of the grid could be a point selected by direct manipulation, with gravity pulling to predefined and non-numerically specified points such as the eight corners of a box, .n, .e, .s, .w, .ne, .se, .sw, and .nw.

To add more direct manipulation will require more creativity to come up with interfaces that constrain it in a natural and unobtusive way to that which corresponds to what humans want. That is the challenge for the designer of the next version.

or after which to apply parameters required pulling down and clicking a menu item rather than simply pointing at the object with the mouse. Thus, to him, WD-pic leaves much to be desired in direct manipulation. He found it much less painful to just edit the internal **Pic** representation to add the paramters or new objects anywhere, and got into the habit of using WD-pic to rapidly produce a first version and then to use his favorite text editor for all subsequent changes.

7 Conclusions and Future Work

WD-pic has met some of its goals but not all of them. It certainly is interchangeably batch and WYSIWYG. It certainly produces human-like **Pic** code as its internal representation. The user can certainly produce diagrams that are simple to express in **Pic** much much faster with WD-pic than with other systems, as these require simply a series of button pushes and typing of simple text in text windows and no movement of the mouse to the drawing canvas.

The program that was produced has the right design except for the lexical portion and the use of standard widgets, and the user interface is clumsy in the less used parts of the language.

The problems in the user interface are not fixable given the decision to use standard widgets. These widgets require bringing up an interaction window when it is desired to input tokens from the key board and these windows require confirmation of the input. The profusion of pop-up windows that present themselves when one begins to have parameters and deviate from the defaults is inundating. Each one takes time to fill and demands confirmation. It would be much much nicer to be able to simply type what needs to be typed directly to a window that comes up under where the cursor is pointing without having to wait for it to come to the screen and for the mouse to point to it and without having to confirm what has been typed. This would also make it easy to use the mouse in place of textual description of places. Of course doing so means using an entirely different set of user-interface widgets, ones that seem to differ from the standard that seems to have established itself among most applications running on X, Windows, and MacOS. It may even be necessary to program it from scratch.

The decision to use standard widgets, wrong in retrospect, was taken to make it easy to prototype, i.e., to rapidly explore different options and be able to throw out what has been done already without feeling that a lot of work has been wasted. This is the first time, in the author's advisor's experience, that the requirements of the prototyping process prevented adopting good requirements for the software.

The other main deficiency is in direct manipulation. It would be nice to be able to place a box where the cursor points rather than at the default position or a position specified by explicitly writing positioning parameters either via pop up window or via the editor in the internal representation. The problem with these direct manipulations is that they get in the way of adhering to the goal of keeping the internal representation similar to what a human would write. If one can place a box anywhere on the canvas, one will find **Pic** code like

box at 1.234297, 5.8593683

in the internal representation.

6. It is impossible to delete simultaneously a number of independent elements from the picture by direct mouse manipulation.

7. It is impossible to print the displayed picture, directly from WD-pic.

It follows from the author's experience using WD-pic

1. that it is preferable to construct a picture sequentially, according to the order of element in the picture,

2. that it is desirable to understand clearly connections between elements before starting to draw the picture and to use the possibilities of WD-pic, such as placing one element relative to another.

3. that the user must work in the insertion mode with care, especially if he needs to change directions in the process of insertion, because the initial part of a picture influences the rest and it could lead to many difficulties for complex insertion. Sometimes it is better to make an insertion using the *Text edit* option of the *Edit* menu.

6.4 Advisor Assessment

Besides the author, the advisor of the author exercised the prototype, although not as extensively as the author. Recall that the advisor was also the customer for the program and was responsible for supplying the requirements for WD-pic. He exercised the prototype for two main purposes:

1. to determine if the prototype captured his intentions.

2. to use it to make some figures that he needed for papers he was writing.

He knew the batch program **Pic** very well, often able to write **Pic** descriptions of figures strictly from imagining the figure. He did not know WD-pic well and, initially had to ask the author how to do certain things.

He reports tremendous satisfaction when using WD-pic to prepare diagrams for which he could either think the description in real time or for which he prepared a hand-written draft of the specification ahead of time. In these cases, things went super fast as he was able to click buttons for the commands in rapid sequence much much faster than typing out the commands with the help of an editor, that is it was much much faster to click the box button, then the arrow button, and then the box button, click, click, click, that it was to type out "box; arrow; box". When a button click had to be accompanied by text entry, such as for the textual contents of a box, the WD-pic was not quite as satisfying, because the entry of text required a dialog window to pop up, the text to be entered, and then that entry to be confirmed by clicking "Yes"; this is considerably slower than just typing the text enclosed by standard double quote marks. Still even less satisfying and in many cases, a down right nuisance were the heavily parameterized commands and changing already entered commands. These involved many many pop-up windows each of which demanded confirmation that it had understood what had just been entered. Moreover, selecting the drawing object to which

Here is the **Pic** text generated by *fig2dev* from the picture, created in *xfig*:

circle at 1.000,9.025 rad 0.255
circle at 2.012,9.012 rad 0.255
line from 1.256,9.012 to 1.756,9.012

From this example, we can see that the **Pic** text generated by WD-pic is more user-friendly. Also the **Pic** text generated by WD-pic can be easily reused in other picture, reusing the text created by *fig2dev* is more difficult, because the coordinates of the circles and of the line's end points are defined explicitly.

- Changing position or size of one element in *xfig* can destroy entire picture. In WD-pic, this destruction can be avoided because connections between elements are realized in the picture's internal representation.

  For example if we want to change the **circles** sizes in the Fig 10 using *xfig* we need to change size of first circle, readjust line, destroy the second **circle**, copy the first **circle**, and readjust the new second **circle**. On the other hand, in WD-pic, we need only to change the size of the first **circle**.

- In WD-pic we can easily insert new elements in the middle of an element chains and easily delete elements from the chain without any changes in the following part. In *xfig* we need to divide the figure into the parts on opposite sides of the insertion point and move at least one of them to make room for the inserted element.

On the other hand, WD-pic, in its present realization, is more limited then *xfig* in drawing complex non-regular pictures. However, once the picture is drawn in WD-pic, it is easier to correct such pictures because of features described above. WD-pic's set of primitive elements is poorer. It also doesn't have the full set of element manipulations that *xfig* has.

We can say that WD-pic has the following shortcoming, independently of **Pic** limitations

1. It is impossible to change sizes of a **box, circle** or **ellipse** by direct mouse manipulation.
2. It is impossible to change position of a **box, circle** or **ellipse** by direct mouse manipulation.
3. It is impossible to insert an entire file into the **Pic** specification by direct mouse manipulation.
4. Editing functions such as Copy and Paste, working with clipboard, and Undo are not available.
5. It is impossible to choose a font for a **text** string.

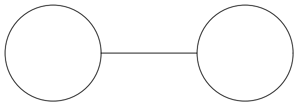

Figure 10: Simple example

Such kind of pictures can be produced simply by clicking push buttons with subsequent adjustment of sizes and insertion **text** without adjustment or placement of elements.

For example the Fig. 10 was produced with the help of WD-pic and *xfig*. In WD-pic the following sequence of actions was executed.

1. Click the push button *circle*.
2. Click the push button *line*.
3. Click the push button *circle*.
4. Click the push button *same*.

In *xfig* the following sequence of actions was executed.

1. Click the push button for **circle**.
2. Adjust the circle size.
3. Click the push button for **line**.
4. Adjust the line length and direction.
5. Click the push button copy.
6. To copy the first **circle** and adjust it position.

In WD-pic for the all elements, we use the default sizes and don't need to adjust sizes or positions of elements .

- Here is the **Pic** text generated by WD-pic for this picture:

circle
line
circle same

```
A15 :box
arrow
A16 :box
arrow
A17 :box
spline from A10 .w left linewid *3 down lineht *1 \
then left linewid *1 down lineht *1 to A15 ->
spline from A12 .w left linewid *1 up lineht *1 \
then left linewid *2 up lineht *1 \
then left linewid *1 down lineht *1 \
then left linewid *1 down lineht *1 to A16 ->
spline from A14 .w left linewid *3 to A17 ->
.PE
```

Fig.11

```
.PS
ellipse width 1 height 0.6 "WD-pic" "interface"
arrow
A1 :circle radius 0.6 dashed "Set of" "callback" "functions"
arrow
box width 0.8 height 0.6 "Data" "structures"
arrow
A0 :box width 0.8 height 0.6
box at A0 "picture"
arrow from A1 .s down lineht *1
A2 :line left 0.50 down 0.30 \
then right 0.50 down 0.30 \
then right 0.50 up 0.30 \
then left 0.50 up 0.30
"Pic text" at A2 +(0 ,-0.3 )
.PE
```

6.3 Author's Assessment

WD-pic was utilized to produce a large number of pictures. Some of them are provided throughout this text.

First of all, it must be mentioned that it is simpler and faster to draw pictures with the help of the WD-pic than to create and correct **Pic** files directly, using standard UNIX text editor such as vi, because we provide all the possibilities for forming **Pic** file but add additional options for direct graphical online editing.

A popular tool for producing line drawing, which is supplied in the X Windows environment is *xfig* drawing editor. From the comparative study of WD-pic and *xfig* following conclusions can be derived.

- For simple linear diagrams such as chains of element (**boxes, circles,** or **ellipse**), possibly containing **text**, and connected by **lines, arrows** or **spline**, for example Fig. 1, Fig. 2, Fig. 5, WD-pic is more comfortable to use than *xfig*. This occurs because by default **Pic** produces chains of drawing elements with default sizes and placement.

Fig.10

```
.PS
down
"Object array"
move down 0.5
A9 :box width 0.6 height 0.3
A11 :box same
A13 :box same
move right 1.50 up moveht *6
down
A10 :box height 0.2 "Pic ptr"
box " ..."
A0 :box height 0.2 "o_nt1"
A2 :box same "o_nt2"
box "..."
move
A12 :box height 0.2 "Pic ptr"
box "..."
A4 :box height 0.2 "o_nt1"
A5 :box same "o_nt2"
box "..."
move
A14 :box height 0.2 "Pic ptr"
box "..."
A7 :box height 0.2 "o_nt1"
A8 :box same "o_nt2"
box "..."
move right 2.00 up moveht *9
down
"String array"
move down 0.30
A1 :box
box
A3 :box
box
box
box
A6 :box
arrow from A0 .e to A1
arrow from A2 .e to A3
arrow from A4 .e to A3
arrow from A5 .e to A6
arrow from A7 .e to A6
arrow from A8 .e to A6
arrow from A9 to A10 .nw
arrow from A11 to A12 .nw
arrow from A13 to A14 .nw
move left 3.00 up moveht *6
down
"Pic Text List"
move down 0.30
```

```
move right 2.00 up moveht *9
down
"String array"
move down 0.30
A1 :box
box
A3 :box
box
box
box
A6 :box
arrow from A0 .e to A1
arrow from A2 .e to A3
arrow from A4 .e to A3
arrow from A5 .e to A6
arrow from A7 .e to A6
arrow from A8 .e to A6
arrow from A9 to A10 .nw
arrow from A11 to A12 .nw
arrow from A13 to A14 .nw
.PE
```

Fig.9

```
.PS
linewid =0.4
arrow "File" above
move left linewid *1 down lineht *1
arrow right linewid *1 "GUI" above
move up 0.24
right
A1 :box width 0.8 height 0.8 "WD-pic"
arrow
B0 :box same "PIC" "compiler"
arrow
A0 :ellipse width 1.2 height 0.8 "PIC" "Data Structures"
arrow
circle "Interface" "to" "TROFF" radius 0.5
arrow
A3 :arrow from A0 .s down lineht *2 \
then left linewid *4 \
to A1 .s
A2 :arrow from A1 .s down lineht *2
ellipse same "WD-pic" "Data Structures"
"1" at A2 +(-0.09 ,-0.53 )
"2" at A3 +(-1.13 ,-0.94 )
box with .nw at B0 +(-0.7 ,0.8 )width 4.3 height 1.5 dotted
.PE
```

```
box width 0.8 height 0.5 "Parameters"
arrow same
box height 0.3 "adjust"
box same "to element"
A9 :ellipse at A7 +(1.2 ,-1 )same dashed "self-standing" "text"
arrow same
box width 0.8 height 0.5 "Parameters"
arrow same
A0 :box height 0.3 "place"
arrow from A7 .s to A8 .n
arrow from A7 .s to A9 .n
box same with .nw at A0 .sw "adjust"
.PE
```

Fig.7

```
.PS
move down 1 right 1.5
box width 1.2 height 0.8 "PIC" "compiler"
arrow
ellipse width 1.2 height 0.8 "PIC" "Data Structures"
arrow
circle radius 0.5 "Interface" "to" "TROFF"
.PE
```

Fig.8

```
.PS
down
"Object array"
move down 0.49
A9 :box width 0.6 height 0.3
A11 :box same
A13 :box same
move right 1.50 up moveht *6
down
A10 :box " ..."
A0 :box height 0.2 "o_nt1"
A2 :box same "o_nt2"
box "..."
move
A12 :box "..."
A4 :box height 0.2 "o_nt1"
A5 :box same "o_nt2"
box "..."
move
A14 :box "..."
A7 :box height 0.2 "o_nt1"
A8 :box same "o_nt2"
box "..."
```

Fig.4

```
.PS
scale =1.1
movewid =0.27
A2 :box width 1 height 0.7 "Pic tokens"
A3 :ellipse at A2 -(2 ,1.9 )same width 0.8 height 0.7 "box" "circle" "ellipse"
move
A4 :ellipse same "line" "spline" "arrow"
move
A5 :ellipse same "move"
move
A6 :ellipse same "arc"
move
A7 :ellipse same "text"
arrow from A2 .sw to A3 .n
arrow from A2 .sw to A4 .n
arrow from A2 .s to A5 .n
arrow from A2 .se to A6 .n
arrow from A2 .se to A7 .n
arrow from A3 .s down 0.30
box width 0.8 height 0.5 "Parameters"
arrow down 0.30
box height 0.3 "sizes"
box same "lines"
box "place" same
box same "fill"
box same "same"
arrow from A4 .s down 0.30
box width 0.8 height 0.5 "Parameters"
arrow down 0.30
box height 0.3 "direction"
box same "line"
box same "from-to"
box same "->"
box same "same"
arrow from A5 .s down 0.30
box width 0.8 height 0.5 "Parameters"
arrow same
box height 0.3 "direction"
box same "to"
box same "same"
arrow from A6 .s same
box width 0.8 height 0.5 "Parameters"
arrow same
box height 0.3 "radius"
box same "invisible"
box same "from-to"
box same "direction"
box same "->"
A8 :ellipse at A7 +(-0.10 ,-1 )width 1 height 0.6 dashed "text " "associated with" "element"
arrow same
```

```
arrow
A1 :ellipse width 0.9 height 0.6 "Pic file"
arrow
box "Pic"
arrow
box "TROFF"
arrow
circle radius 0.4 "Picture"
arrow from A0 .s down lineht *1
circle same "Picture"
arrow from A1 .s
circle same "Disc"
.PE
```

Fig.2

```
.PS
down
ellipse width 1.2 height 0.8 "Event" "generated by GUI"
arrow
box "WD-pic"
arrow
A0 :ellipse width 0.9 height 0.7 "Unique" "number"
right
arrow right linewid from A0 .e
A0 :box width 0.8 height 1 "Pic" "interpreter"
arrow
circle radius 0.6 "drawing" "data structures"
arrow
box width 1 "picture"
arrow from A0 .s down linewid *1
circle same "Pic internal" "representation"
.PE
```

6 Evaluation of usability of WD-pic

6.1 Method

As stated in the introduction, it is desired that the evaluation be as realistic as possible, consisting in using WD-pic to build diagrams that are like those in real-life as much as possible. Accordingly, it was decided to use WD-pic to prepare all the figures in this report. The rationale is that the pictures in this report are typical of those needed in computer science documents, precisely the domain of pic.

The author used WD-pic to draw all of these figures and others that were not used. The author had become an expert in WD-pic by virtue of having written it; she knew exactly what is allowed and what is not and exactly what would do the job and what would not. Thus, she represents the expert user. As mentioned, she used what she learned from doing these drawings to discover problems with the requirements of WD-pic and to suggest alternative behaviors.

The author's advisor used WD-pic to draw a few pictures for papers he was writing. Although he is an expert in **Pic** and can generate **Pic** code for a picture on the fly, he was a total novice in the use of WD-pic and actually encountered quite a few difficulties, some of which led to changes in the software's requirements.

Both compared their efforts to make drawings with WD-pic to their efforts to make the same drawings with *xfig*, which they were both expert at.

This chapter first shows an example of making the same drawing in *xfig* and WD-pic in order to give the reader a flavor of the way these comparisons went. A number of pros and cons of WD-pic are apparent from this example. Next, the WD-pic generated **Pic** code for many of the figures shown in this book is shown. These show that the main goal of human-like **Pic** as the internal representation has been achieved. The author's advisor agrees that he would make **Pic** very much like these if he were specifying them in normal batch **Pic**. Finally comes a list of the drawbacks of WD-pic. These all amount to problems with the current requirements for WD-pic, requirements that will have to change if WD-pic is ever to be a successful software system. It will be seen that many of these come from constraints that were observed to meet the main goals and many of these come from the original choice of using a set of widgets to build the user interface.

6.2 Figures in Thesis

Here are texts of **Pic** files that correspond to pictures presented in this work. Please note how close these are to what a skilled human user of **Pic** would write.

Fig.1

```
.PS
right
A0 :box "WD-pic"
```

- *New scale.*
  The treatment function for this menu item executes following steps:

  1. Create a prompt dialog, allowing the user to insert new value of scale.
  2. Read the new scale value and changes the *pixel* and the *inch* fields for all element of the default table.
  3. Replace the old scale value by the new one.
  4. Add a new element, containing a pointer to the string "scale = *new_scale_value*", to the **Pic** text list.
  5. If the current mode is *insert*, then the second method of graphic input treatment is executed.

- *New Default.*
  The treatment function for this item executes following steps:

  1. Create a dialog, containing a list of all default parameters and their current values. The user can change these current values.
  2. For each default parameters, check if it was changed. If a parameter was changed, make all necessary changes in its default table element, corresponding to the new value.
  3. For each changed default parameter, add to the **Pic** text list a new element, containing a pointer to the string "*default value name = new_default_value*".
  4. If the current mode is *insert*, then the second method of graphic input treatment is executed.

- *Reset.*
  The treatment function for this menu item executes following steps:

  1. Find all default table elements for which the current default value is not equal to the **Pic** default value.
  2. Create a dialog, containing the names of all such elements. This dialog allows resetting either some of their values or all of them.
  3. For each reset parameter, replace its *inch* value by its *def_inch* value and recalculate its *pixel* value, using screen metrics.
  4. If any of parameters were reset, for all such parameters, add to the **Pic** text list a new element containing a pointer to the string "reset *default value name*". If all parameters were reset add to the **Pic** text list a new element containing a pointer to the string "reset all".
  5. If the current mode is *insert*, then the second method of graphic input treatment is executed.

5.2.10 *Structures* popup menu callback function

The callback for the *Structure* popup menu chooses functions for treatment of *loop* and *if... then...else* items.

- The treatment function for *loop*

  1. creates a dialog, which allows the user to insert the loop's index, the high and low boundaries and the step.
  2. creates a temporary file and writes the **Pic** text corresponding to the current picture into the file and adds the string "from *index-name* = *low-boundary* to *high boundary* by *step -name* do ..." to the current insertion point of the file.
  3. executes the same steps as for the *Text edit* item of the *Edit* menu.

- . The treatment function for *if ... then ... else*

  1. creates a temporary file and writes the **Pic** text corresponding to the current picture into the file and adds the string "if ... then ... else to the current insertion point of the file.
  2. executes the same steps as for the *Text edit* item of the *Edit* menu.

5.2.11 *Defaults* popup menu callback function

The *Defaults* popup menu callback function chooses a treatment function according to the selected menu item. All information related to the **Pic** default sizes is contained in the special default table, containing all **Pic** default values. Each table element has the following structure:

char name[11];
int pixel;
double inch;
double def_inch

Here

- *name* - is the name of the default parameter.
- *pixel* - is the current parameter value in pixels.
- *inch* - is the current parameter value in inches.
- *def_inch* - is the **Pic** default value for the parameter in inches.

The items of the *Default* popup menu allow the user to change the value of any parameter.

- *Modify attribute*: The treatment function for this item creates a dialog, containing a text window with a **Pic** string describing the selected object. The pointer to this string is contained in the current **Pic** text list element. After the user has finished changing the string, the pointer to the new string replaces the pointer to the old one in the current **Pic** text list element.

- *Delete attribute*: The treatment function for this item deletes element of the **Pic** text list corresponding to the current drawing object. After that it gets a new data structures from the **Pic** compiler, as was described for the second method of graphical input treatment. Finally, it redraws the picture.

- *Text edit*: The treatment function for this item:

    1. creates a temporary file named tempXX.pic, where XX is the current process number. Using the **Pic** text list, it writes into this file the text in **Pic** format corresponding to the current picture, and after that closes this file.

    2. creates a child process with the help of the *fork* system call, and waits for its completion.

    3. the child process in turn invokes the text editor named by the environment variable EDITOR with the help of the *exec* system call.

    4. after the user has finished changing tempXX.pic and has closed the editor, the parent process continues its execution and invokes the **Pic** compiler as described earlier, with standard input redirected from the temporary file.

    5. after getting new data structures, the picture is redrawn, and the temporary file is deleted.

5.2.9 *View* popup menu callback function

The *View* popup menu contains only one item, *View*. The callback for this item

1. creates a prompt dialog, in order to get a file name.

2. if the user didn't insert file name, all strings from **Pic** text list elements are copied into a one character array in the memory with addition of special symbols to the beginning and to the end. If the user inserted a file name into the prompt dialog, the file is read into a character array.

3. creates a scrolled text, containing a **Pic** file.

4. destroys the temporary file, if one was created.

1. Search for the element in the object array whose screen coverage includes the coordinates of the point.

2. The selected object array element is a pointer to the structure that contains the field with the address of the corresponding **Pic** text list element. The special pointer to the current element of the **Pic** text list, *curPicText*, is established pointing to this **Pic** text list element. The special pointer to the previous element of **Pic** text list is established pointing to the previous element of the current **Pic** text list element.

3. The selected element is redrawn with a double line by the function *DrawSelectElement*.

Here is code for element selection:

```
/* (x, y) are coordinates of the point, selected with the help of mouse */
x = event- >x;
y = event- >y;
/* Searching an element containing the point (x,y) */
sel_obj = FindObj(x,y);
if (sel_obj!= NULL)
{
/* curPicText is a pointer to the Pic text list element, corresponding to the chosen object */
                curPicText = sel_obj- >PicTextUnion.Ptr;
/* prevPicText is a pointer to the Pic text list element, previous to the element, pointed by curPicText
*/
                prevPicText = DefinePrev(curPicText);
                IsSelect = 1;
/* redrawing of the selected element by double line */
                DrawSelectElement(sel_obj);
                CurElemNum = DefineElemNum(sel_obj); }
```

- *Insert*: The treatment function for this item establishes *insert* mode.
- *Add*: The treatment function for this item:

  1. establishes *insert* mode, and
  2. reestablishes the previous pointer to the current pointer, and the current pointer to the next element of the current **Pic** text list element.

- *End Insert—Add*: The treatment function for this item cancels *insert* mode.

- The treatment function for the menu item *Store as New File* is:

  1. A file selection dialog is created in order to give the user the possibility to choose a file name.

  2. The **Pic** text is recorded into the file the same way as for the *Save Current File* item.

- The treatment function for the menu item *Read File* is

  1. Destroy the **Pic** text list, object array, and string array.

  2. Clear the drawing area.

  3. A file selection dialog is created in order to give the user the possibility to choose a file.

  4. Using the *fork* and *exec* UNIX system calls, create a child process to execute the **Pic** compiler.

  5. With the help of a *pipe*, establish a connection between WD-pic and the child process.

  6. The standard input of the **Pic** compiler is redirected from the chosen file.

  7. **Pic** compiler begins its execution.

  8. When the **Pic** compiler finishes its execution, it sends new data structures to WD-pic using the *pipe*.

  We can see that this method looks like the second method of graphic input treatment, except that **Pic** compiler's input is redirected from the file and not from the *pipe*.

- The treatment function for the menu item *Quit* quits the WD-pic execution with the help of the UNIX system call *exit*

5.2.8 *Edit* popup menu callback function

Edit popup menu callback function chooses a treatment function according to the selected menu item.

- *Select element*: The treatment function for this item establishes the handler function for a mouse press event in the drawing area. In order to select a picture element the user must press the left mouse button at some point of the drawing area belonging to the element. Pressing the mouse button invokes the handler function, which executes the following steps:

Below is the list of parameter groups and parameters, which are treated by the second method in the *append* mode:

1. Sizes group
2. Sources and destinations group
3. Places group, except the **at** parameter for placing element by its center
4. Arc directions group.
5. **same** parameter

5.2.6 Drawing directions treatment

The treatment function for drawing directions executes the following steps:

1. Define a current mode (*append* or *insert*).
2. Add to the **Pic** text list a new element containing a pointer to the string with the chosen direction name.
3. In *append* mode, give to the global variable Direction a new value.
4. In *insert* mode, execute the second method of the graphic input treatment.

5.2.7 *File* popup menu callback function

This callback function chooses a treatment function for the selected menu item.

- The treatment function for the menu item *New* is:

    1. Destroy the **Pic** text list, object array, and string array.
    2. Clear the drawing area.
    3. Create the empty object array and the empty string array.
    4. Assign the default values.

- The treatment function for the menu item *Save Current File* is:

  If the **Pic** file, corresponding to the picture has no name yet, i.e., the picture was not read from the file and was not saved before, then the treatment function for the *Store as new File* item is called.
  Otherwise, starting from the beginning of the **Pic** text list, strings in **Pic** format from each element of the list are written to the file. Of course, to the beginning and to the end of the file, the special **Pic** symbols, .PS and .PE, are added.

1. Add this **text** to the string of the **Pic** text list element corresponding to the **Pic** element containing this **text**.

2. If there is a free place in the string array, insert into it the new element. If string array is full its length is increased by a constant number and the first free place is used for the new element.

3. Change field, *o_nt2*, in the object structure corresponding to the **Pic** element with which the string is associated.

- For self-standing **text**.

1. Add a new element to the **Pic** text list.

2. Insert a new element into the string array, as was described earlier.

3. Create a new object structure in the object array the same way as was done for the other **Pic** tokens.

For *insert* mode, the treatment function just changes the **Pic** text list as was described before and uses the **Pic** compiler to get new data structures. After updating or receiving new data structures from the **Pic** compiler, the treatment function for **text** executes following steps:

1. geometric calculation based on the object array

2. drawing

5.2.5 Element parameter treatment

Earlier, we defined two methods of treatment of input data, invoked with the help of the graphic interface. In the classification of the **Pic** language (Chapter 3), almost all element parameters were divided into groups. Now, parameters from these groups and parameters, which are not united into groups can be divided according to their treatment method in the *append* mode. Below is the list of parameter groups and parameters that are treated by the first method in the *append* mode:

1. Line Styles group

2. Directions group

3. Arrow Heads group

4. Adjustment group

5. **fill** parameter

6. **at** parameter from the Places group, when the user wants to place an element by the coordinates of its center

```
ptr- >o_type = BOX;
ptr- >o_nt1 = ptr- >o_nt2 = NumText;
ptr- >o_mode = CurDirection;
ptr- >o_count = 2;
ptr- >o_val[0] = MyDefaultTable[0].pixel;
ptr- >o_val[1] = MyDefaultTable[1].pixel;
ptr- >o_attr = 0;
ptr- >o_ddval = MyDefaultTable[14].pixel;
ptr- >o_fillval = MyDefaultTable[18].pixel;
/* Addition of the new element, containing pointer to the string "box" to the Pic text list */
CreatePicElementText(!ElemReference,"box",BOX);
/* Establishing pointer to the corresponding Pic text list element */

ptr- >PicTextUnion.Ptr = curPicText;
```

- For *insert* mode, we use the second method of graphic input treatment:

  1. The new element is inserted into the **Pic** text list. This element contains a pointer to the string in the **Pic** language corresponding to the **Pic** token.

  2. The second method of data treatment, described above, is executed.

After either updating the old data structures or getting new data structures, the treatment function

- executes all necessary geometric calculations, using the object array.

- draws the resulting picture.

The treatment function for **move** executes all those steps, except last two in the normal mode (geometric calculation and drawing).

The treatment function for **text**, different from other **Pic** token treatment functions, needs additional information. In order to get this information, it creates a standard prompt dialog in which user is asked to insert a string.

There are two kinds of **text** in the **Pic** language:

1. **text**, associated with a **Pic** element.

2. self-standing **text**.

For each kind of a **text**, there is a different treatment in the *append* mode.

- For **text** associated with a **Pic** element:

1. If there is a free place in the object array, it is used as a pointer to the structure, corresponding to the new object. If the object array is full, its length is increased by a constant number, and the first free place is used as a pointer to this structure.

2. The structure corresponding to the new object is filled according to **Pic** defaults and the current state of the picture.

3. A new element, containing a string in **Pic** format that corresponds to the **Pic** token is added to the **Pic** text list and the pointer to the new element of the **Pic** text list is inserted into the object structure.

Here is some code from the treatment function for **box**. This part corresponds to the *append* mode execution. This code creates and fills the object structure, that was described in the previous section.

```
obj* ptr;
ElemReference = 0;
/* In function Create Obj free place searching in object array and object structure creation are executed */
ptr = CreateObj(2);
/* CurrentX and CurrentY are coordinates of the current insertion point */
ptr- >o_x = CurrentX;
ptr- >o_y = CurrentY;
/* Calculations of coordinates of the box center and coordinates of the new current insertion point. These calculation depend on the current picture direction */
if (CurDirection == R_DIR) {
ptr- >o_x += 0.5 * MyDefaultTable[0].pixel;
CurrentX += MyDefaultTable[0].pixel;
}
if (CurDirection == L_DIR) {
ptr- >o_x -= 0.5 * MyDefaultTable[0].pixel;
CurrentX -= MyDefaultTable[0].pixel;
}
if (CurDirection == U_DIR) {
ptr- >o_y -= 0.5 * MyDefaultTable[1].pixel;
CurrentY -= MyDefaultTable[1].pixel;
}
if (CurDirection == D_DIR) {
ptr- >o_y += 0.5 * MyDefaultTable[1].pixel;
CurrentY += MyDefaultTable[1].pixel;
}
/* Structure filling */
```

1. WD-pic, creates a empty character array in memory . All strings from Pic text list are copied into this array. Also special **Pic** symbols are added to the beginning and end of the array. At the end, this array contains the **Pic** text corresponding to the picture on the screen.

2. WD-pic, using *fork* and *exec* UNIX system calls, creates a child process, which executes the **Pic** compiler.

3. With the help of a *pipe*, WD-pic establishes a connection with its child process.

4. The standard input of **Pic** compiler is redirected to the *pipe*.

5. Using the *pipe*, WD-pic delivers the string to the **Pic** compiler as an input and waits for data structures produced by **Pic**.

6. The **Pic** compiler starts execution.

7. When the **Pic** compiler finishes its execution it sends the new data structures to WD-pic using the *pipe*

5.2.4 Pic token treatment

Here is a part of function **Interpreter** code.

```
case (_BOX) :
/* Call of treatment function for box */
                BoxFunction();
/* If push button parameter set for box exists then break */
                if (currentPanelNumber == 0) break;
/* If exists push button parameter set for any Pic
token except box then destroy it */
                if (currentPanelNumber != -1)
                        XtDestroyWidget(buttonFrameParam);
/* Create push button set for box */
                BoxParameters();
                currentPanelNumber = prevPanelNumber = 0;
                break;
```

This code calls the treatment function for the **Pic** token **box** and creates the push button parameter set for **box**. For other **Pic** tokens, the same operations are executed.

The treatment functions for a **box, line, arrow, ellipse, circle, arc,** and **spline** first of all check the current mode, *append* or *insert*, and then execute the following steps:

- For *append* mode, they update the data structures using first method of graphic input treatment. This method for **Pic** tokens is explained in detail below.

Button Function, StructCb and **DefaultCb** correspond to **Pic** language elements. **FileCb, EditCb** and **ViewCb** correspond to interactions of the graphic interface with the file and the current picture.

5.2.2 Push buttons Callback function - ButtonFunction

All push buttons, described in the interface part, share one and the same callback function, called **ButtonFunction**. Each push button has a unique number. After the user has pushed chosen push button, **Button Function** starts its execution with the push button's unique number as a parameter.

ButtonFunction executes in two steps:

1. It returns the background and the foreground of the previously selected push button, (if present) to the original and reverses the background and the foreground of the currently selected push button.
2. It calls function **Interpreter** with the push button's unique number as a parameter.

The function **Interpreter**, according to its parameter, chooses the proper treatment function for the selected push button.

5.2.3 General description of push buttons treatment functions.

All treatment functions translate events, invoked by the user interface, into a string or substring of **Pic** lexical tokens, which must be added to the **Pic** picture representation. Some of these functions do not need additional information, for example treatment functions corresponding to **box, move,** or **circle** push buttons. Some of them do need additional information, for example, the treatment function corresponding to the **text** push button. Such functions create a new elements of interface in order to get this information. Each treatment function executes in two modes:

- *Append* mode, corresponding to an addition of a new string or substring to the end of the **Pic** picture representation.
- *Insert* mode, corresponding to an insertion of a new string or substring in the **Pic** picture representation.

In Subsection 5.1, we described two methods of treatment of input data, invoked with the help of graphic interface. All treatment functions can be divided into two groups:

- Callback functions, which implement in *append* mode the first method.
- Callback functions, which implement in *append* mode the second method.

In *insert* mode all treatment functions implement the second method. Realization of the first method doesn't introduce any problems, but realization of the second method is more complex and needs detailed explanation:

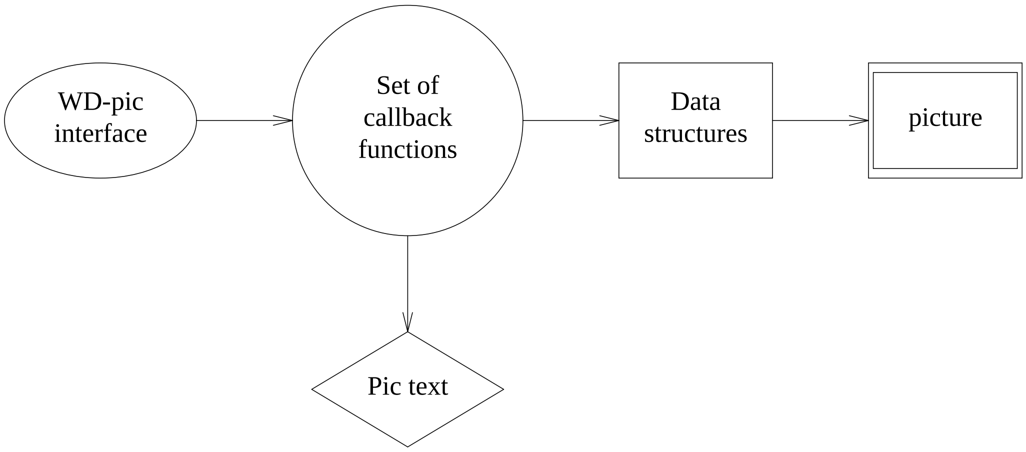

Figure 9: Connection between the interface and either data structures or **Pic** text

5.1.7 Summary

Using **Pic** compiler data structures, we created the WD-pic data structures, allowing us to store all necessary information for working with a picture via both graphical interface and a text editor interface. After that, we modified the **Pic** compiler data structures in order that they conform to the WD-pic data structures. Also we changed the **Pic** compiler code so that it supports the modified data structures.

5.2 Internal program organization

5.2.1 Callback function as a connection between the graphic interface and data structures

In previous sections, we described the interface of WD-pic and the picture representations in WD-pic. The user can change the picture with the help of elements of the interface via data structures. Also, using elements of the interface, he/she can deal with the file, containing the **Pic** text corresponding to the picture. In Motif, each element of interface has a set of events, each of which can be treated by a callback function. We can consider callback functions, corresponding to each element of the WD-pic interface as a connection between the graphic interface and either the data structures or the **Pic** file (Fig. 9).

In the table below, the callback functions for all elements of the interface are listed:

Elements of the interface	**Callback functions**
Push buttons	**ButtonFunction**
File popup menu	**FileCb**
Edit popup menu	**EditCb**
View popup menu	**ViewCb**
Defaults popup menu	**DefaultCb**
Struct popup menu	**StructCb**

header.

```
***********************************************************************/
/***********************************************************************/
picText is a pointer to the beginning of Pic text array.
curNumPicText is the index of the current Pic Text Array element.
***********************************************************************/
if (NewElem) {
/********** New element or structure header ***********/
curNumPicText++;
if (curNumPicText >= maxNumPicText) {
picText = (PicTextDef *) grow((char *)picText,
"PicTextDef", maxNumPicText += 1000,
sizeof(PicTextDef));
}
picText[curNumPicText].curString =
(char*)malloc(length * sizeof(char));
strncpy(picText[curNumPicText].curString, text, length);
picText[curNumPicText].curStringLength = length;
}
else {
/********** Element parameter or body of a structure ************/
picText[curNumPicText].curString =
(char*)realloc(picText[curNumPicText].curString,
(picText[curNumPicText].curStringLength += length) * sizeof(char));
ptr = picText[curNumPicText].curString +
(picText[curNumPicText].curStringLength - length) * sizeof(char);
strncpy(ptr, text, length);
}
free(text);
}
```

In the YACC code of the **Pic** compiler for ELLIPSE is the definition

```
prim:     ELLIPSE attrlist     { $$ = circgen($1); }
```

In the function circgen(), new elements are added to the object array and the string array. In order to establish connection between the object array and the Pic text array, the new string was added to the function:

```
p->PicTextUnion.Num = curNumPicText;
```

Here, p is a pointer to the object array element corresponding to the ellipse.

Pic text list. Because these functions are very important, we give here the code of one of them, which is used in **Pic** compiler.

This function gets as input:
textElement, a **Pic** word,
Element, its code,
NewElem, a parameter that marks a **Pic** element that is not inside a **Pic** operator, a **box, arrow, circle, ellipse, spline, move,** self-standing **text**, or the first word of a **Pic** operator (**for** or **if**).

For a **Pic** word that is marked by *NewElem*, we begin a new element in the **Pic** text array. For all other **Pic** words, we just add a new word to the string of the current **Pic** text array element.

```
void CreatePicElementText(NewElem, textElem,Element)
int NewElem;
char* textElem;
int Element;
{
int length;
char *ptr;
char *text;
int i;

    /* If textElem contains a full Pic word we add a blank to the end of the word */
    if (Element != 999)
{

    length = strlen(textElem) + 1;
    text = malloc(length * sizeof(char));
    strncpy(text,textElem,length -1);
    text[length-1] = ' ';
    }
    else

    /* If textElem contains only one symbol, figure or letter, which
is part of number or label, we don't add a blank */
    {
    length = 1;
    text = malloc(sizeof(char));
    text[0] = textElem[0];
    }
    /*****************************************************************
    There are two possible cases :

    1. a new element in the picture or the header of a structure i.e., an if
       or a loop, that requires a new element addition to the Pic text array

    2. An existing element parameter or the body of a structure that
       must be added to the string of Pic text array element. This string
       already contains a Pic element description or a Pic structure
```

In this subsection we are using italic font for code, which was added to the **Pic** compiler, and normal font for its original code.

A new array was added, the **Pic** text array. Its elements have the same structure as an element of the **Pic** text list of WD-pic, and contain the same information. The code below contains a description of the **Pic** text array element structure:

```
typedef struct Def{
            char *curString;
            int curStringLength;
} PicTextDef;
```

The following variables were added to the **Pic** compiler's code:

```
    /* a pointer to the beginning of the Pic text array */
PicTextDef *picText;

    /* the index of the current element of Pic text array */
int curNumPicText;

    /* the maximum number of elements in the Pic text array */
int maxNumPicText;
```

Here is the initialization of the Pic text array:

```
picText = (PicTextDef *) grow((char *)picText, "PicTextDef",
            maxNumPicText += 1000, sizeof(PicTextDef));
```

For compatibility with WD-pic, the new field, *PicTextUnion*, was added to the object array. This field contains an index of the corresponding element of Pic text array.

Besides the changes in the data structures, some additions were made to the **Pic** compiler's code. The main additions were in the lexical analysis part. For each **Pic** word, function *CreatePicElementText* was added. This function builds the **Pic** text array. For example, for the word *ellipse* we have now the LEX code:

```
<A>ellipse      { if (MarkStruct ! = String_pic){
                 CreatePicElementText(!ElemRef,"ellipse",ELLIPSE);
                 ElemRef = 1;
                 }
                 return(yylval.i = ELLIPSE);
                 }
```

The function *CreatePicElementText* adds a new element to the **Pic** text array. This element contains a pointer to the string *ellipse*. This function makes all necessary changes to the **Pic** text array. In WD-pic, the analogous function makes all necessary changes to the

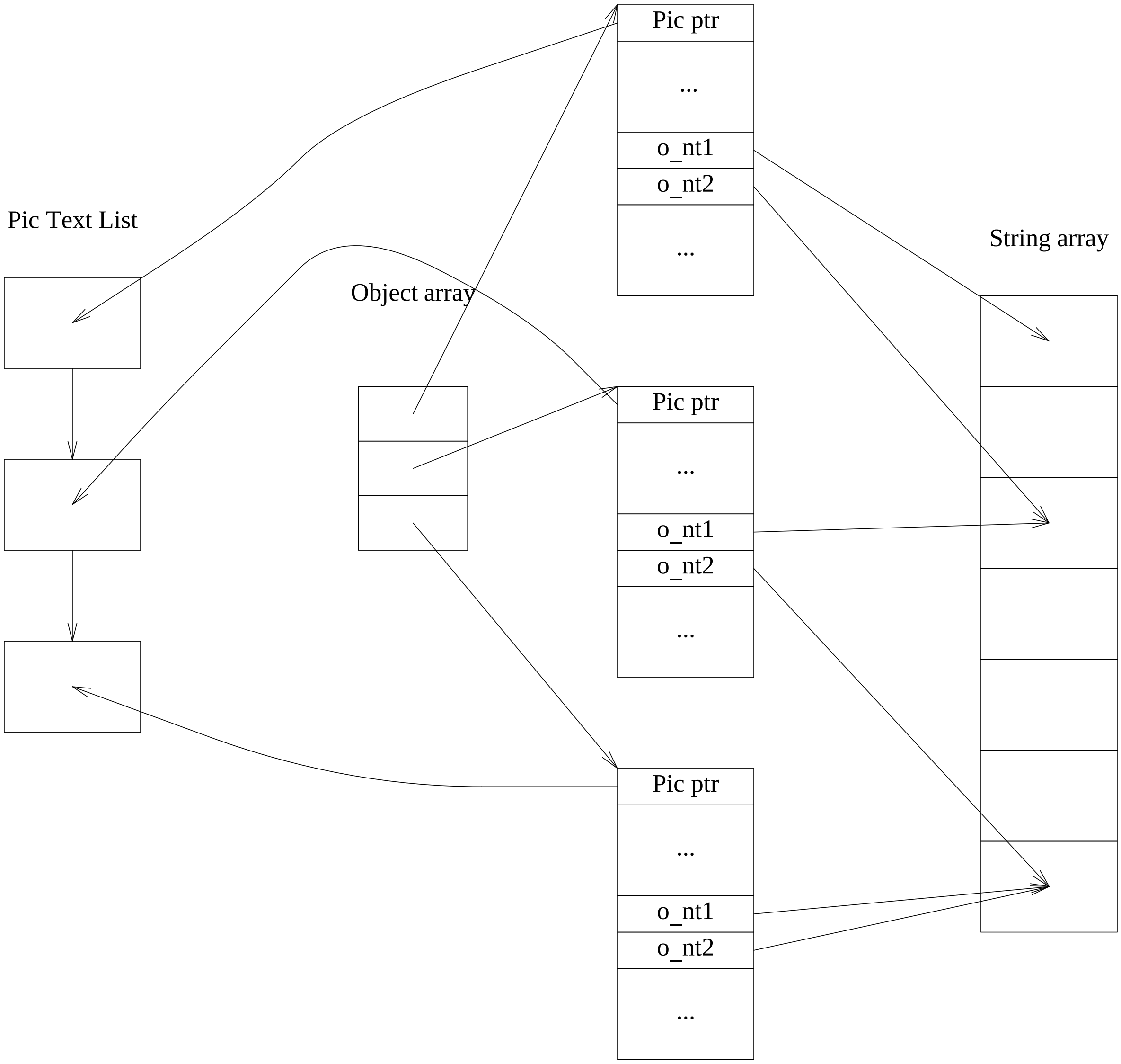

Figure 8: Connections between the **Pic** text list, the object array and the string array

necessary data structures and passes them to the WD-pic for it to draw the picture. This passing is indicated by the arrow marked "2" in Fig. 7.

- user data, created with the help of the graphic interface. In this case, there are two types of input data treatment:

  1. For the first type, WD-pic either creates new data structure elements or adds new information to existing ones. This data flow is indicated by the arrow marked "1" in Fig. 7.

  2. For the second type, created by the interface, a picture representation in the **Pic** language is used. WD-pic changes only this representation and uses it to create a file according to the **Pic** standard. This file is passed to the **Pic** compiler as input data. The **Pic** compiler creates all necessary data structures and passes them to the WD-pic for it to draw the picture. This passing is indicated by the arrow marked "2" in Fig. 7. This method is used, when complex calculations are needed, for example, to change **box, circle,** or **ellipse** sizes, to place an element, etc.

5.1.4 Pic data structures

An internal picture representation in the **Pic** language is stored in the memory as a linked list, called the Pic text list. This data structure allows carrying out such operations, as an insertion of a new element into the list and a deletion of an element from the list. Appendix B contains a detailed description of Pic text list

Each element of the **Pic** text list corresponds either to a **Pic** element, to a **Pic** operator, or to a **Pic** default size change.

5.1.5 Drawing data structures

As was mentioned earlier, the drawing data structures were borrowed with some additions from the **Pic** compiler. In the **Pic** compiler, all information necessary for drawing is contained in the elements of the object array. We use this object array in WD-pic. However, in order to keep a connection between an element's drawing information and its description in the **Pic** language, a new data field was added to the structure of the object array element. This data field allows accessing to the corresponding element of the **Pic** text list. The detailed description of object array is shown in Appendix B.

In Fig. 8, we can see connections between the **Pic** text list, the object array, and the string array.

WD-pic also uses the string array. Each element of the string array contains the drawing information for its **text**

5.1.6 Changes to the Pic compiler's code

Because we want the **Pic** compiler to create the data structures that can be easily transformed to WD-pic data structures, some changes were made to the original **Pic** compiler.

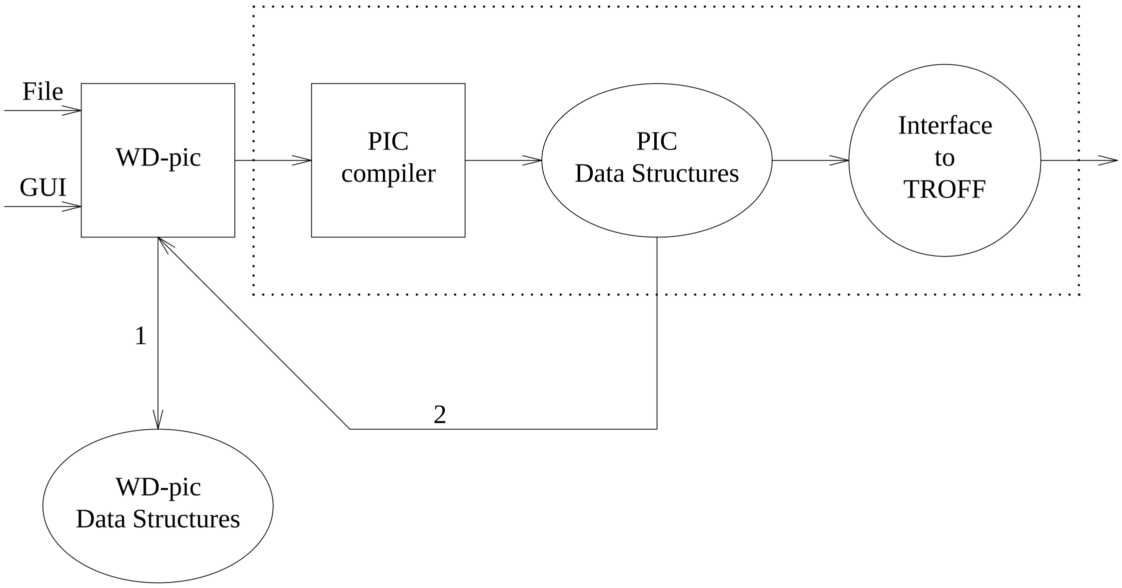

Figure 7: WD-pic execution

first text string which does not belong to the **Pic** element corresponding to the object structure.

- String array. Each element of the string array corresponds to a **text**, which may be associated with a picture element or may be self standing, and also contains the **text**'s adjustment (left, right, up, down) type.

5.1.2 Methods of picture representation in WD-pic

A picture is represented in WD-pic in two different ways:

- with the help of drawing data structures, structures containing all necessary geometric and drawing information for each element of the picture.

- with the help of a data structure containing a representation of the **Pic** text, "**Pic** data structure" for brevity.

Of course, a connection exists between the drawing data structures and the **Pic** data structure.

5.1.3 WD-pic execution scheme and its data structures

In Fig. 7, we can see the scheme of a WD-pic execution. WD-pic has two kinds of input data:

- data structures, created by the **Pic** compiler.

  We have this data when the user wants to read a text file in the **Pic** language. Then this file is passed to the **Pic** compiler as input. In turn, the **Pic** compiler creates all

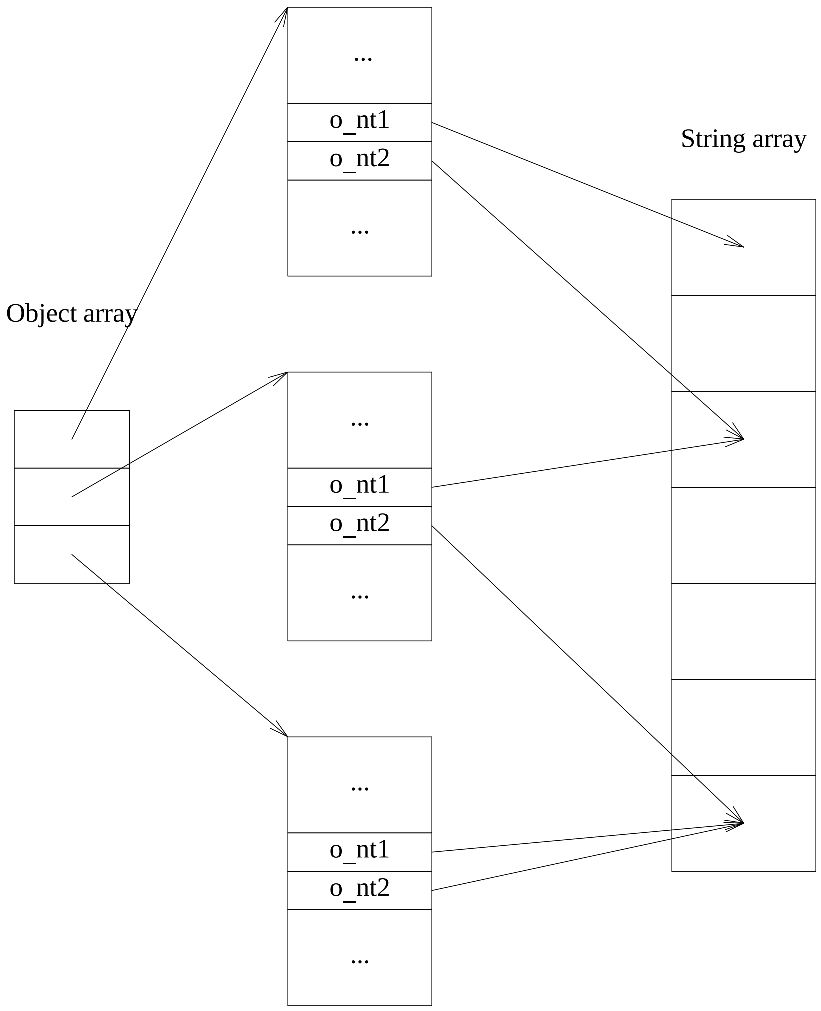

Figure 6: Connection between object array and string array

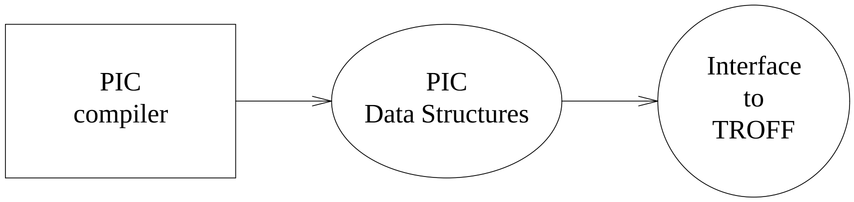

Figure 5: Connection and data flow between **Pic** and TROFF

5 Implementation

5.1 Data structures

5.1.1 Pic compiler execution and its data structures

In order to understand the behavior of WD-pic, we need to analyze the execution of the **Pic** compiler and the data structures created by **Pic** compiler.

In Fig. 5, the connection between and the data flow between **Pic** and TROFF is presented.

Schematically, we can represent a run of the **Pic** compiler as the following sequence of steps:

- The **Pic** compiler gets a **Pic** file, i.e. program, written in the **Pic** language, describing a picture, as input.
- Using lexical and syntax analysis, the **Pic** compiler translates the input into special data structures, containing all the necessary geometric and drawing information.
- Using these data structures, the **Pic** compiler produces a description of the same picture in the TROFF language.

It seems natural to use the data structures, created by **Pic**, with some changes, in WD-pic. It gives the possibility to simplify the interface of WD-pic with the **Pic** compiler and allows, in some cases, to use the **Pic** compiler directly for creating drawing data structures, as will be described later.

Fig. 6 presents two data structures of the **Pic** compiler, the object array and the string array.

- Object array - an array of picture elements. Each element in the object array contains a pointer to an object structure. Each object structure contains an element type (box, circle, etc.), an element geometry, a line style, a fill mode, etc. Also an object structure contains two indexes of elements in the string array. The first, *o_nt1*, is the index of the element, containing a first text string, that belongs to the **Pic** element corresponding to the object structure. The second, *o_nt2*, is the index of the element containing the

4.5.1 *File* popup menu

The *File* popup menu has the following items: *New, Save Current File, Store as New File, Read File,* and *Quit.* These options allows the user to start a new picture, to save a picture, to save a picture with a new name, to read a picture, and to finish work with WD-pic.

4.5.2 *Edit* popup menu

The *Edit* popup menu has the following items: *Select element, Insert, Add, End Insert/Add, Modify attribute, Delete*, and *Text edit.* These options allows the user to select any element of a picture, to add a new group of elements before or after the selected element, to finish new element addition, to modify an attributes of the selected element, to delete the selected element, and to change a picture useing text editor.

4.5.3 *View* popup menu

The popup menu *View* contains only one item: *File View.*, which allows the user to view any existing **Pic** file and also the **Pic** text corresponding to the current picture.

4.5.4 *Structures* popup menu

WD-pic provides the possibility of using **Pic if** statements and **for** loops. These items were regarded as not really appropriate or feasible for direct manipulation as they represent repeatedly conditionally performed requirements of button clicks. Therefore it was decided to use a menu interface for them. The *Structures* popup menu contains two options, *Loop* and *if ... then ... else.* for **if** statements and **for** loops accordingly.

4.5.5 *Defaults* popup menu

WD-pic provides scale and default sizes control. For this purpose, the popup menu *Defaults* contains three items, *Scale, New defaults,* and *Reset.*

4.6 Conclusions

The elements of the user interface follow the structure of the language elements described earlier in Section 3. This assists in the prototyping effort in that whenever there is a change to the structure of the language or to the way dealing with an element of the language, the place in the interpreter affected by the change is clear.

4.4.1 Size control

For each kind of **Pic** basic element, except **text**, of course there is a push button in the corresponding push button set, that allows controlling the element's size.

For a **box, circle,** or **ellipse** this is the **size** push button. The size of a**line, arrow,** or **spline** can be changed with the help of the **direction** push button. For an **arc**, the **radius** push allows to change a size of element. For a **move**, this is the **place** push button.

4.4.2 Place control

As for size control, for place control there is also a push button that allows controlling the element's placement.

For a **box, circle,** and **ellipse**, it is the **place** push button. For an **arrow, line, spline,** or **arc**, there is a **from-to** push button, which allows placing the beginning of the **arrow, line, spline** or **arc**, and its end.

4.4.3 Line style

The **lines** push button serves for selecting a line's style. There are four possible line styles: **solid**, **dotted**, **dashed** and **invisible**.

4.4.4 Fill style

In **Pic**, the attribute **fill** applies only to **box**, **circle** and **ellipse**.

4.4.5 Attribute same

The push button set for each element except **text** and **arc** contains the push button **same**. This push button serves for changing the size of the current element according to the size of last element of the same kind.

4.5 Menu bar

The menu bar contains five popup menus:

- *File*
- *Edit*
- *View*
- *Structures*
- *Defaults*

The menu bar can be conditionally divided into two parts. The first part, containing the *File, Edit,* and *View* popup menus, corresponds to the interaction of the graphic interface with the file and the displayed picture. The second part, containing the *Structures* and *Defaults* popup menus, corresponds to the operator group and default group of **Pic** language elements.

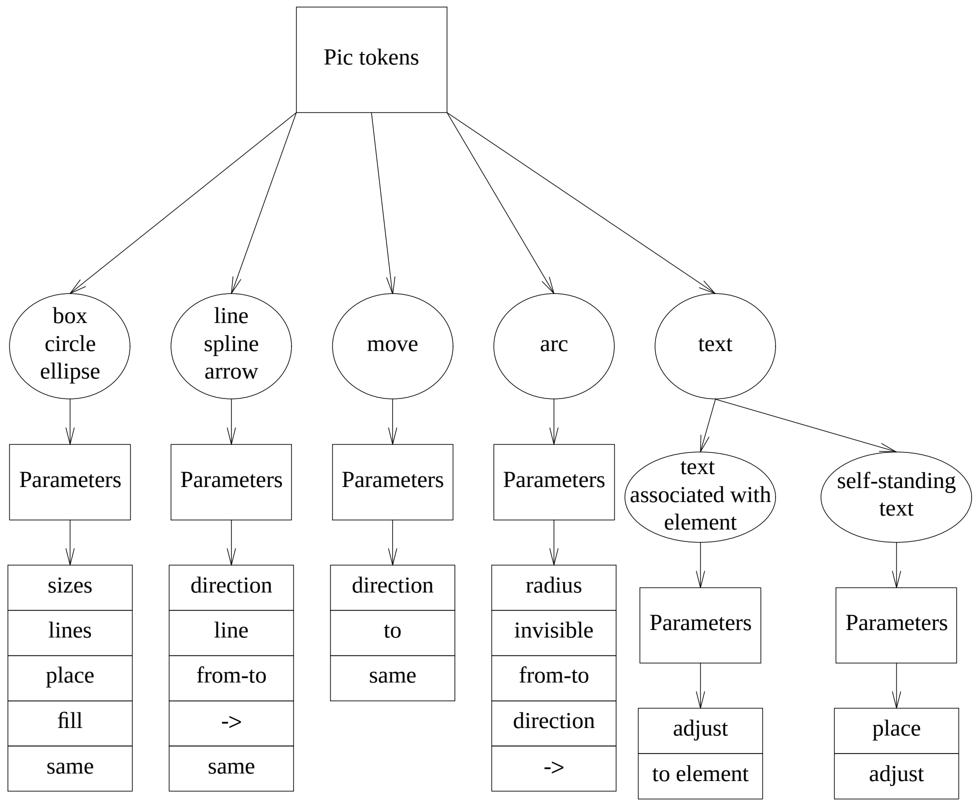

Figure 4: **Pic** tokens and their parameters.

2. **line** - to change the element's line style.
3. **from - to** - to change the beginning and the end of the element.
4. $->$ - to put an arrowhead on the element .
5. **same** - the element has the same size as the previous element of this kind.

The parameters for **arc** are:

1. **radius** - to change the element's radius.
2. **invisible** - to make the element invisible.
3. **from - to**- to change the beginning and the end of the element.
4. **direction** - to choose the direction of the element, which may be either counterclockwise or clockwise.
5. $->$ to put an arrowhead on the element.

The parameters for **move** are:

1. **direction** - to change the direction of the element.
2. **to** - to choose the end point of the element, which may be relative to another element of the picture.
3. **same** - the element has the same size as the previous element of this kind.

The parameters for **text** associated with an element are:

1. **adjust** - to specify the adjustment of the *text* relative to its position.
2. **to element** - to return to the push button set, corresponding to the element, containing this *text*.

Text associated with an element is the only kind of **Pic** element that can be embedded in another one. In order to allow the user to continue to work with the enclosing **Pic** element, the parameter **to element** was introduced.

The parameters for self-standing **text** are:

1. **place** - to choose the *text's* position.
2. **adjust** - to specify the adjustment of the *text* relative to its position.

In Fig. 4, a structure showing which parameters appear for each **Pic** token is presented.

4.4 Drawing element parameters

Element parameters allow the user to change the size of an element, its placement, or its line style, to fill a **box**, **circle** or **ellipse**, to change an **arc**'s direction, to put an arrowhead on a **line, spline, arrow,** or **arc**, or to specify the adjustment of **text**. There are default values for all these parameters in the **Pic** language. For example, the default line style for all elements is solid, the only placement is at the current insertion point, the default fill value for **box, circle**, or **ellipse** is 0.3, and the default **arc** direction is counterclockwise. If any parameter is not given for any element, the default applies. In the table below, the size names and their default values are presented.

description	parameter	value(inches)
width of a **box**	**boxwid**	0.75
height of a **box**	**boxht**	0.5
width of an **ellipse**	**ellipsewid**	0.75
height of an **ellipse**	**ellipseht**	0.5
radius of a **circle**	**circlerad**	0.25
radius of an **arc**	**arc**	0.25
horizontal **line** length	**linewid**	0.5
vertical **line** length	**lineht**	0.5
horizontal **move** length	**movewid**	0.5
vertical **move** length	**moveht**	0.5

All defaults are chosen so that many typical diagrams that computer scientists make in technical papers can be produced entirely with elements without any parameters. This means that in WD-pic, these diagrams can be constructed almost entirely by clicking element buttons one after the other, using the keyboard only for **text**.

In the beginning, the drawing element parameters part of the button panel is empty. After the user selects an element from the **Pic** token part, then a set of push buttons appears, each of which allows the user to define a different drawing parameter of the current element. The parameters for **box, circle,** and **ellipse** are:

1. **sizes** - to change the element's size.
2. **lines** - to change the element's line style.
3. **place** - to change the element's place.
4. **fill** - to fill the element.
5. **same** - the element has the same size as the previous element of this kind.

The parameters for **line, arrow,** and **spline** are:

1. **direction** - to change the element's direction.

4 User Interface

4.1 Main Window

WD-pic starts up with the main window shown in Fig.3.
There are three parts in the main window:

- the button panel in the left part of the main window.
- the menu bar
- the drawing area with vertical and horizontal scrollbars, used to reveal additional parts of the drawing area

The button panel is divided into three parts in accordance with the **Pic** language classification that was given earlier:

- **Pic** tokens
- drawing directions
- element parameters

Each of these parts contains buttons, in principle, for each of the elements of these classifications described above. However, due to space limitations, at any time, only these elements that are legal to choose at this time are displayed. This, of course, helps the user to avoid what would be a syntax error in the **Pic** language.

4.2 Pic tokens

The **Pic** tokens part of the button panel contains ten push buttons; one push button for each **Pic** basic element; **box, line, arrow, circle, ellipse, arc, spline, move, text**, and ";". The last is used to mark the end of the current element description.

4.3 Drawing directions

In **Pic**, there are four drawing directions, **up, down, left, right**, which control the direction of drawing from the point in which they are pushed. The buttons for directions contain arrows indicating the direction. Accordingly, these buttons are called arrow buttons.

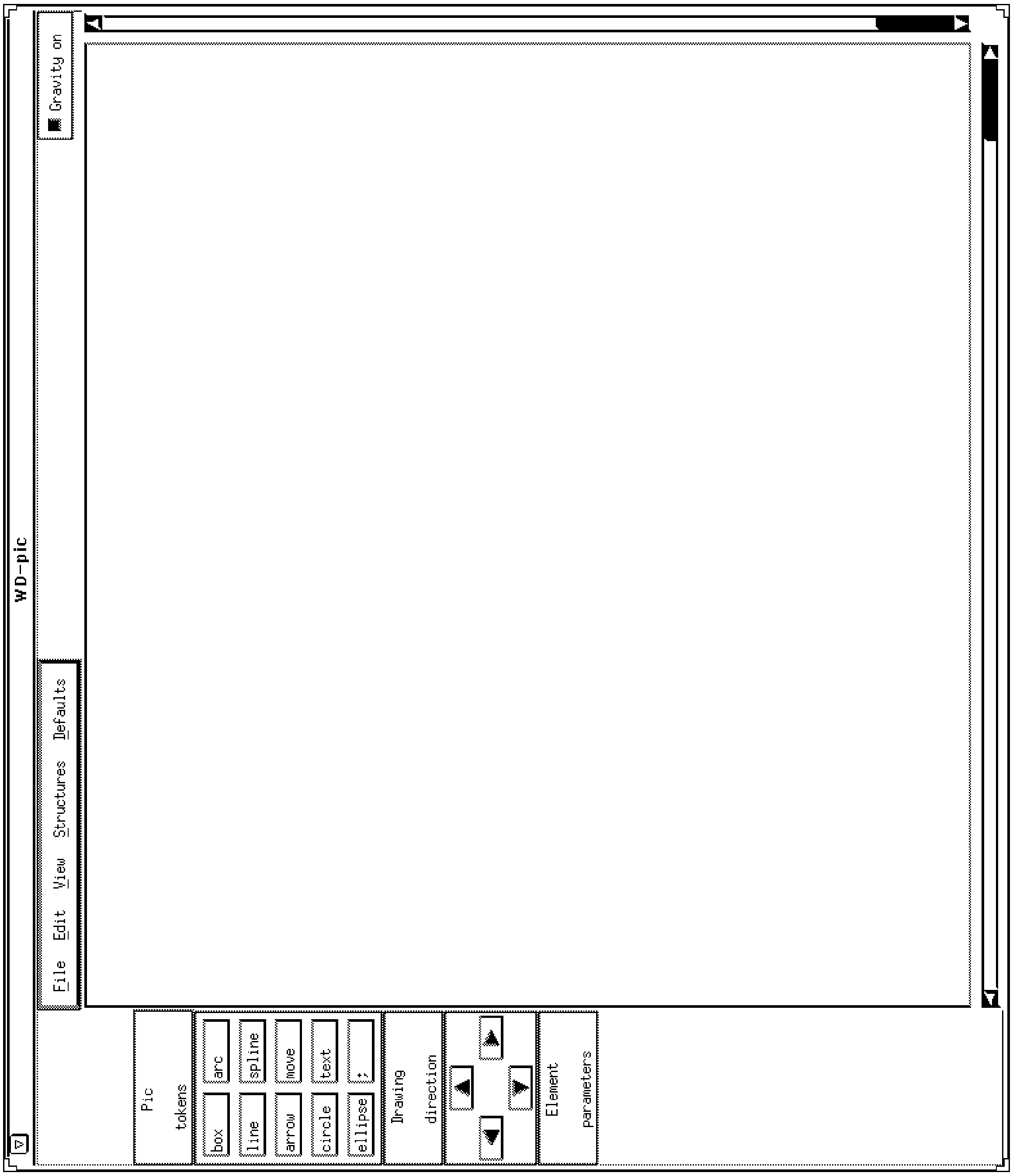

Figure 3: MainWindow of WD-pic.

- Element parameters: some of them can be united into groups.

Group	**Pic elements**
Sizes	**width, height**
	radius, diameter
Directions	**up, down, right, left**
Line style	**solid, dotted, dashed, invisible**
Sources and destinations	**from, to**
Places	**at, with ... at**
Arrow heads	$->, <-, <->$
Adjustments	**above, below, right, left**
Arc directions	**clockwise**

  Other element parameters are **fill** and **same**

- Default values: **scale, boxwid, boxht, linewid, lineht, circlerad, arcrad, ellipsewid, ellipseht, movewid** and **moveht**

- Operators: **if ... then ... else ...** and
  for ... to ... by ... do ...

3.4 Conclusions

All of the issues described above affect the user interface, which is the subject of the next chapter, and ultimately the requirements of the program under consideration.

1. When the user draws a picture on a screen using the graphic interface, he/she simultaneously creates the picture's internal representation in the **Pic** language. This internal representation not only saves the geometric and drawing characteristics of a picture's elements, such as coordinates of center, sizes, line style, etc., but also the relationship between elements. The graphic interface allows placing one element relative to another, specifying the size of one element relative to that of another, etc.

2. When the user changes the **Pic** file corresponding to a picture with the help of a text editor, he/she immediately sees a changed picture on the screen. After that the user can continue to work with the picture using the graphic interface. Note that, the WD-pic interface allows passing freely from WYSIWYG picture drawing to editing of the internal representation and vice versa.

At any given time, the picture on the screen is the same as the picture created via **Pic**, TROFF, etc. with the accumulated internal representation as input. For each picture element, all its parameters, for example size, line style, etc., can be defined explicitly with the help of the WD-pic interface. If a parameter is not defined, then it is assigned the corresponding **Pic** default value. Each new element is placed always at the current insertion point, which is by default, the end point of the last element. WD-pic allows placing the element in any place in a picture, using explicit instruction, that redefine the current insertion point while describing a new element. Simultaneously, the description of this element in **Pic** format is added to the current insertion point in the internal representation. The current insertion point has a visual manifestation as well as being the place in the internal representation after which the new tokens go. At any time, this accumulated internal representation (AIR) can be converted into a **Pic** file by adding special symbols to the beginning and to the end. After the insertion of a new element, the picture displayed on the screen is the same as that generated on paper by submitting the AIR to **Pic** piped to TROFF, with the geographic center of the two pictures being the same.

3.3 Conditional classification of Pic language elements

It is convenient for our purposes to divide the main elements of the **Pic** language into five groups:

- **Pic** tokens: **box, circle, ellipse, line, spline, arrow, move, text, arc**, and the special symbol ";".

- Drawing directions: parameters, that influence the rest of the picture. There are four such parameters, **up, down, left,** and **right**.

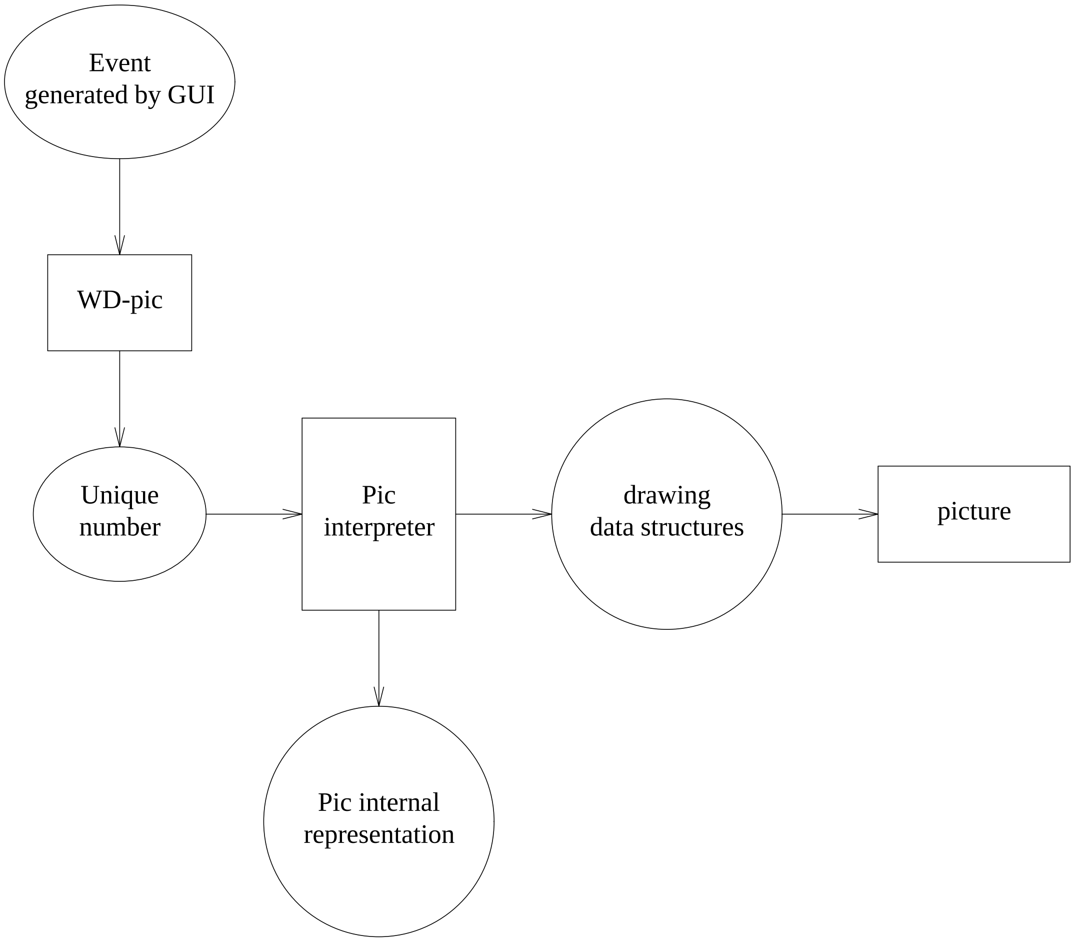

Figure 2: Principal execution steps in WD-pic from GUI event to the picture

in a default position or as the next item in the current path, as is the default with both the **Pic** internal representation of *Picasso* and in the PostScript internal representation of of the canvas drawing. Moreover, when one looks at the **Pic** or PostScript generated for these internal represention or the **Pic** or PostScript obtained by fig2dev from an *xfig* internal representation, one sees lots and lots of multi-digit real numbers as x and y values, even if all items in fact are of default size and placement. Moreover, the **Pic** code generated for a box is four lines connecting the corners expressed as a pair of multi-digit real number.

Therefore, it is fair to say that no existing system meets the stated initial and key requirements for WD-pic.

3 High Level Design

3.1 Basic Design Idea

In WYSIWYG drawing software, which is used in Graphical User Interface (GUI) environments, it is possible to make all screen drawing with the help of the mouse and to use keyboard input to insert text. Each action with the mouse or keyboard is called an event. The principal execution steps in WD-pic in going from a GUI event to the picture on the screen are shown in Fig.2.

1. WD-pic gives to each kind of event, invoked by the user interface, a unique number, called a token.
2. The **Pic** interpreter belonging to WD-pic transforms the unique number into drawing data structures and simultaneously into an internal representation in the the **Pic** language. When the **Pic** compiler translates **Pic** text into TROFF format, it creates data structures, for producing text in TROFF format. When we run the **Pic** compiler on **Pic** text, created by WD-pic, we get drawing data structures. These data structures are equivalent to the drawing data structures created by WD-pic itself.
3. The drawing data structures are used for drawing the picture on the screen. Of course, this picture is the same as the picture created by **Pic**, TROFF, etc. given as input the **Pic** text produced by WD-pic.

3.2 Two ways of creating picture and its internal representation in the WD-pic

There are two ways to create a picture and its representation in WD-pic:

- using the WYSIWYG graphic interface to draw the picture,
- writing a textual description of the picture in the **Pic** language.

From the user's point of view, these two ways are equivalent in the final effect.

will not be in a form interpretable by the program, i.e., the resulting specification is not well formed. The user should always save a back-up copy to allow restoring to an interpretable specification should the editing hopelessly mess up the specification. Note also that none of these programs were designed on the assumption that the user would be manually editing the saved file; hence they take no pains to insure clarity of the representation. *MacDraw*, *Paint*, and *PowerPoint* all produce binary saved files to protect the manuafacturers' trade secrets. For the systems whose saved representation is not PostScript and is therefore not directly printable, there is a way to generate a PostScript description of the saved figure. In all the programs, in the printable PostScript representation, either the saved file itself or one obtained by translation from it, the constant numbers defining positions and sizes are multi-digit numbers with long fractional parts, e.g., 2.079573454, decidedly not what a human being would choose to write, e.g. 2 or even 1.25. Moreover, in the case of *Paint*, the PostScript is that of a bitmap rather than a sequence of commands to draw the figure.

2.3 Conclusions

All of the reviewed programs are really only either batch or WYSIWYG. None are both.

The closest that comes to being both is *Picasso*, which is intended to be a WYSIWYG version of **Pic**. However, it does not allow editing of the internal pic from within *Picasso*. If one edits it externally, there is the problem of assuring that the result is syntactically and semantically correct. *Picasso* simply breaks when it attempts to set itself up on an internal representation that is not syntactically or semantically correct.

The same problem holds for *xfig* and *AI*, whose internal representations are fairly obscure, but ASCII files.

The internal representation of *xfig* has for each object a line consisting of a list of numbers, the first being the object type, and the rest being its parameters, the last of which are the defining ordered x,y pairs. It is hard to edit this internal representation to yield another meaningful, *xfig* interpretable internal representation and a good picture. Note that the author's advisor does this editing all the time to effect changes that are a pain to do with the standard interface, e.g., change all occurence of a particular word to another or to change all rectangles into circles, but he constantly backs up internal representation files in case he makes errors that are too messy to undo.

The internal representation of *AI* is a rather obscure PostScript that interprets to a picture only in the presence of a prolog defining the abbreviated instructions that are used in the internal representation. Thus, it is hard for the user to read, understand, and modify the internal representation in a manner that yields syntactically and semantically correct PostScript and a good picture. The same holds for *idraw*'s internal representation. It is claimed by the authors of both *AI* and *idraw*, that any PostScript program can be converted into a form readable by the program to allow imported PostScript programs to be manipulated in a WYSIWYG fashion.

Every single one of the WYSIWYG programs known to this author requires placement of objects in the canvas. That is when the box is requested by clicking on the box button, the mouse must be used to position the box in the canvas. The user must click the mouse on two points or click on one corner and drag to the opposite corner. None put the box

Of course, all of these have all the drawbacks of batch programs and none of the advantages of the WYSIWYG, direct manipulation programs.

2.2 WYSIWYG

There are a large variety of WYSIWYG, direct manipulation drawing programs. It seems that every operating system builder has one and there are still others supplied by third parties. This section reviews a representative collection in a more tabular fashion. They are

1. *Adobe Illustrator (AI)*
2. *idraw*
3. *MacDraw*
4. *PageDraw*
5. *Paint*
6. *PowerPoint*
7. *xfig*

AI is produced by Adobe and it runs on all three major platforms, PCs with Windows, Macintoshes with MacOS, and Suns with UNIX. *idraw* is a public domain program distributed in source form and can be run on any platform with a C compiler. *MacDraw* is produced by Apple and runs on Macintoshes with MacOS. *PageDraw* is another public domain program that runs on PCs with Windows. *Paint* is produced by Microsoft and runs on PCs with Windows. *PowerPoint* is produced by Microsoft and runs on PCs with Windows. Actually *PowerPoint* is a general formatting system for making slide shows and has a picture drawing palette to enable drawing pictures for these slides. *xfig* is a public domain program distributed in source form and can be run on any platform with a C compiler and X windows, as it assumes the X protocol for accessing the graphic screen on which to draw the pictures.

All of them have a very similar user interface with a palette of drawing figures, it seems, always on the left side of the screen and an associated drawing canvas. The palettes all provide basically the same primitive drawing items, including boxes, lines, circles, ellipses, etc. The more advanced of these, such as *AI* and *xfig*, provide an additional collection of buttons that provide a very complete set of locally and globally applicable manipulation commands such as scaling, rotating, translating, etc.

In all of them, after clicking a primitive item, e.g., box, it is necessary to explicitly place and size the item with the mouse on the drawing canvas. None provide default positioning and sizing such as provided by **Pic**.

The saved files for *AI*, *idraw*, *PageDraw*, and *xfig* are in plain ASCII format, with that of the first three being PostScript and that of the fourth being a representation found in no other program. All of these formats are editable by the user, although in the opinion of the author's advisor, only that of *xfig* is really intuitive to the user with commands one-for-one with visible objects on the canvas. While all are editable, the user faces a danger that after editing, it

examples, as opposed to toy, concocted examples, assures a good exercising of the problem domain that is to be covered by the software. Furthermore, these same examples give a way to evaluate the software against real-life uses.

Requirements exploration prototyping requires being able to rapidly put together a new version of the program with maximum reuse of what has been done before and is not discarded. Consequently, some time was spent to identify a syntax directed structure for the basic interpreter. In addition, to make it easy to change the content of the user interface decisions, i.e., what is printed in a prompt to the user and what is expected from the user as a response and as confirmation, it was decided to use a prepackaged library of user-interface widgets. While this choice did make it relatively easy to change the program as requirements were changed, it did end up standing in the way of meeting one of the goals, namely that of interchangeability of batch-style, textual and direct manipulation inputs.

2 Existing Systems

The existing drawing systems come in two different flavors, batch and WYSIWYG. A number of each of these are described in this chapter. It will be seen that none of them meet the requirements set out for WD-pic. First it is clear that any program which is either batch or WYSIWYG cannot meet the requirements of being interchangeably batch and WYSIWYG. Therefore the criticisms are focused on such issues as whether batch-like defaults are available in a WYSIWYG system, whether its internal representation is accessible, understandable, and editable by the user, and how close the internal representation of a picture is to what a human would write to get the same figure in the same language.

2.1 Batch

Two of the main batch drawing programs are **Pic** and *ideal*, both AT&T products. Both are troff preprocessors. pic is an imperative system in which one gives explicit commands for a sequence of figure items to be drawn, one after the other. ideal is a declarative system in which one describes what is in the picture and give equations whose solutions provide values for the constants needed to place and size figure items. Both output troff commands which explicitly draw the figure in terms of points, lines, ellipses, and splines, which are the basic troff drawing commands. A third program is *MetaPost*, a METAFONT to PostScript translator. METAFONT is a combination imperative, declarative system normally used to specify glyphs for a font. Since a glyph is just a drawing, clearly one can use METAFONT to specify arbitrary drawings. The output of *MetaPost* is a sequence of PostScript commands that draw the specified figure.

Of the three, for the kinds of figures that are normally drawn in computer science technical literature, **Pic** has the simplest interface, because it has exactly the right set of primitives and defaults that a flow diagram the sequence of processes P1, P2, and P3 can be specified by the **Pic** commands

"box "P1"; arrow; box "P2"; arrow; box "P3" ".

No other batch drawing program has such a simple specification of the flow diagram.

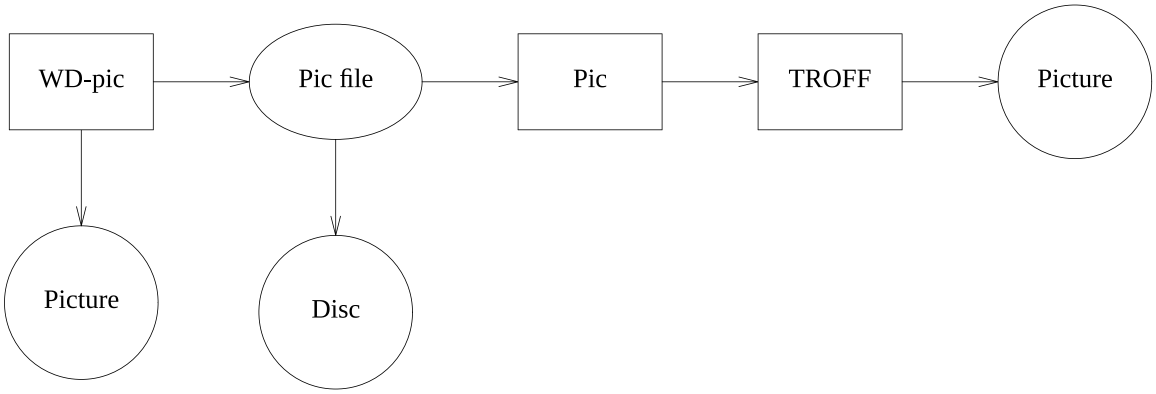

Figure 1: Connection between WD-pic, **Pic** and TROFF.

In this work we tried to make an internal representation of a picture close to what a human would write as possible. For example, if the user wants to draw a line, connecting a specific corners of two boxes, he/she would write something like “line from first box.ne to second box.sw”, which is more understandable then “line from 1.00245 to 3.52671”. In order to follow this goal, we sometimes need to constrain the WD-pic interface.

WD-pic is a UNIX system line drawing program. It is intended chiefly for technical document illustration, such as flow charts, state diagrams and so on. Pictures, interactively created by WD-pic can be saved as batch **Pic** programs, and batch **Pic** programs can be displayed and edited interactively with the help of WD-pic.

In Fig.1, we see the connection between WD-pic, **Pic**, and TROFF.

1.3 Prototiping Method

This thesis is intended to report on an example of prototyping to determine actual requirements starting from a vague idea of the general requirements.

The mission was to decide the requirements and design of an interchangeable batch-WYSIWYG drawing system based on the picture desciption language for the batch program **Pic**. This is a case in which the requirements and design affect each other in that what is not implementable is not requirable (and of course, vice versa). The hard part of this mission is to simultaneously determine a reasonable way to structure the software that would meet the requirements and to determine the best user interface that makes the switching between batch and WYSIWYG input as seamless as possible.

The method used to refine the requirements and to get a first implementation of WD-pic is the traditional requirements exploration prototyping method. A prototype is rapidly programmed to a first version of the requirements. Users get to play with the prototype to discover inappropriate requirements. The requirements are adjusted and a new round begins. The process continues until satisfactory requirements and an implementing prototype are obtained or until it is determined that the path of decisions taken cannot yield satisfactory requirements without throwing out most of the work done so far.

In the case of word-processing software such as WD-pic, the prototypes can be exercised to produce the figures that will be used in the documentation of the software. Indeed, all of the figures in this thesis were produced by the author using WD-pic. Having a set of real-life

1 General Description

1.1 Batch vs. WYSIWYG Drawing Programs

Picture drawing software comes in two different varieties:

- batch, such as **Pic**, and
- WYSIWYG (What You See Is What You Get) direct manipulation, such as *xfig*, *idraw*, *MacDraw*, etc.

The advantages of batch drawing software are:

- The user can insert or delete an element or change its size or location without destroying the entire picture.
- It is easier to manipulate large groups of elements treated as a unit.

The disadvantages of batch drawing software are:

- The user cannot see the picture as it is being composed, and because of it he/she could have problems with proportions, sizes, text string lengths, etc.
- It is easier to make an error.

The advantages of WYSIWYG, direct-manipulation drawing software are:

- The user can see the whole picture.
- It is easier to make decisions about such matters as exact types of elements, for example **box** or **circle**, element sizes, positions of text string, etc., and to avoid obvious errors.

The disadvantages of WYSIWYG, direct-manipulation drawing software are:

- A change of one element can destroy an entire picture.
- It is not comfortable to manipulate large groups of elements, for example changing the default size for all **boxes** or changing the direction of elements in the picture.

1.2 Goal and Requirements for WD-pic

We want to get the best of both worlds in the form of a combination batch, WYSIWYG, direct-manipulation drawing program, called WD-pic (**W**YSIWYG, **D**irect-manipulation **Pic** drawing program), an enhancement of the well-known batch **Pic** program that works with the same **Pic** language[1,2,4]. The **Pic** language is ideally suited for the kinds of diagrams occuring in Computer Science documents. That is it should be possible to get a box drawn by either clicking box button or by typing "box". However, even in direct manipulation, the batch defaults should hold. The internal representation of any picture is an editable **Pic** language description of the picture in the style that a human would enter.

Abstract

The **Pic** language is a TROFF preprocessor for drawing line drawing in UNIX environments. In this work, a classification of the elements of the **Pic** language is introduced, and a convenient graphical interface, corresponding to this classification, WD-pic, for **Pic** users is developed in the X Windows environment, using Motif. WD-pic combines graphics and text building and editing of pictures, and it produces pictures on a screen and a corresponding internal representation, which is fully compatible with the **Pic** language. The displayed pictures in WD-pic are realized partly with the help of the existing **Pic** compiler and partly with new code. The original **Pic** compiler's source code was modified and incorporated as a separate process in WD-pic. Creating pictures in WD-pic was tested and compared with writing description of the same pictures in the **Pic** language and creating the same pictures with the help of *xfig*.

June 26, 1997

List of Tables

List of Figures

Contents