FLEXOR TECHNOLOGY SERIES

FlexOr Frame Structure

2006 June

Gunnar Gotshalks

Table of Contents

1 Introduction

This document describes the structure of a FlexOr Frame based on HTML/XML type of tagging structure. Many of the standard HTML tags are used with similar meanings as in HTML documents. Additional tags are introduced to meet the needs of the FlexOr processes. This is the XML−like part, although the syntax is less rigid than in XML to make frames easier to tag and read.

2 The global structure

Shows the tags with comments written in Postscript style using "% ..."

<FLEXOR> % Start of document
<Prologue for the frame 2.1>
<BODY> % Start of document contents
<Sections : the contents of the document 2.2>
</BODY> % End of document contents
</FLEXOR> % End of document

2.1 Prologue for the frame

The prolog for a frame defines, using tag parameters and Postscript commands, global processing parameters and definitions for the title page and heading line information for the document.

The following example is the prolog for this document.

```
<FLEXOR tocLevel=0 maxline=100 levelmap=2>
  % Set general FlexOr parameters.
  % tocLevel = n prints table of contents for section levels 0 … n.
  % If tocLevel is not used, then there is no table of contents.
  % levelmap = n sets the base section level to n and current level to 0.
  % An actual section level for typesetting is baseLevel + sectionLevel.
  % maxline = ccc sets the number of characters per line for fill.  This
  % is only an approximation that works most of the time.  If the
  % occasional line gets to long, a <BR> tag can be inserted.
```

```
<INCLUDE path=/cs/dept/www/java/bin/FlexOr/frames
           files=FlexOrPs.header file=MathSymbols.header>
  % Files to include as is.  In this case various Postscript definitions.
  % The path is in effect for all included files until the path is changed
  % in the same or later include tag.

<PS>
  % Start Postscript mode.  Until the end tag output as is.
  % Usually all that need be done is to edit the "obvious" strings within ().

% Define the custom constants.

/PageNumber 2 def
/Date (2006 June) def                       % Date for title.
/HeaderDate (2006 June 27) def              % Header date on each page
/HeadTitle (FlexOr Frame Structure) def     % Header line title on each page
                                            % Can be changed using Postscript mode
                                            % later in the document

% Title page information


/Helvetica-Bold findfont 12 scalefont setfont
               % Fontname and size are the parameters on this line
(FLEXOR TECHNOLOGY SERIES) centerText
               % Also have rightJustifyText
16 vtab (FlexOr Frame Structure) centerText
               % "nn vtab" moves vertically nn points down the page

/Helvetica findfont 10 scalefont setfont
16 vtab  Date centerText
16 vtab  (Gunnar Gotshalks) centerText
20 vtab

%-------------------------------------------------------------------------------
% Output the frame itself.
</PS>          % Exit from Postscript mode.
```

2.2 Sections : the contents of the document

After the <BODY> tag the rest of the document is a sequence of sections.

Each section begins with a <SECTION>Name of the section</SECTION> tag. For example this section, the one you are currently reading, has the following section tag.

<SECTION>Sections: the contents of the document</SECTION>

The section tag has a level attribute that is used to set the level of the section and all succeeding sections until the level is changed in another section tag. Changing the level of a section affects the font size and type used to display the section header. Section level numbering has the pattern 1.1.1. Level 0 is the leftmost section number.

For example the following level attribute sets the level of the section to 3. It will use the

font size for `level = levelmap + 3`; where levelmap is defined in a FLEXOR tag or is 0 by default.

<SECTION level=3>Name of the section</SECTION>

3 Text modes

Within a section, text is partitioned into a sequence of five disjoint modes (they are not nested within each other) representing different semantics. Each text mode has its own start tag and optional end tag. The assumption is that the mode remains in effect until either the corresponding end tag is reached, whereupon the mode reverts to documentation mode, or another tag occurs that changes to the mode defined by the tag.

The default mode on entry to a section is documentation mode.

3.1 Documentation mode

Write your documentation in English. Include encapsulated Postscript diagrams using the <IMG> tag. See the following section for instructions on how to include diagrams.

<Including encapsulated Postscript diagrams 7>

The mode tags are <P> ... </P>.

Text is filled from succeeding lines until the maximum output line length is reached. Lines are ragged right (no justification).

3.2 Postscript mode

The contents are emitted to the Postscript output without further interpretation or processing. This enables you to insert any Postscript command into a document; which is how the prologue is handled. For a summary of the more useful Postscript commands, see the following section.

<List of some useful Postscript commands 6>

The header files give a complete definition of the Postscript commands. If you know Postscript you can create your own custom header files to typeset FlexOr frames in any desired style.
The mode tags are <PS> ... </PS>.

3.3 Program mode

The contents are considered to be program text. It is output in program text font and the left margin is indented. Otherwise the contents are output as they are typed including new lines. If a valid language is selected (see tag attributes), then keywords in that language are output in a bold −face font.

The tags <REF>...</REF> tag are used within program mode as their main purpose is to support literate programming with its references to subparts of a program. Although the section reference facility can be used in general as illustrated in this document; in documentation mode, however, it takes additional Postscript tags to get good looking spacing.
The mode tags are <PT> ... </PT>.

3.4 Math mode

The contents are scanned for math entities and math keywords, with the remaining text output as is in math font. New lines in the output occur where they appear in the document. See the following section for more information about writing mathematics in FlexOr frames.

<Using mathematical notation 9>

The mode tags are the following:

- <M> ... </M> – are used to introduce short mathematical expressions within documention text. Such as the expression $A \leq B$ within this sentence.
- <MB> ... </MB> – are used to introduce blocks of mathematical text between paragraphs. The end tag is required.

3.5 Z mode

The mode tags are **<Z> ... </Z>**. The end tag is required.

Z mode is used by the "Z check" process, which extracts all of the text between <Z> and </Z> tags from the input file and stores the result in the file *.szc by default.

The command to use the Z type checker on Prism is `fuzz-Z-typecheck frame-name.szc`. If there are errors in the file, the offending line number and a comment are output to the terminal. Sufficient extra newlines are output to the *.szc file so the line number for a line of Z text in the input file corresponds with the line number in the *.szc file, so the error can be easily found and corrected in the source *.sfx file.

The following are vagaries of the fuzz checker.

- Global variables must be defined within :axdef. … :eaxdef. If there is no constraint the left hand vertical line is not drawn.
- If you use functions that you define, then you must suffix the name with ~. Functions pre−defined in Z do not have the ~ suffix. For example, if you have defined the function double, which multiplies its argument by 2, you would type the use of double as follows. Note that the ~ is not shown in weaved output, as the ~ is only needed for the fuzz type−checker.
  <Z> double~ 4 = 8 </Z>
- When errors are reported always check the names of variables and schemas to the left of the reported error location correspond to defined names.
- For the symbol $\upharpoonright$, use **&sproj** for schema projection, and use **&sres;** for sequence restriction (filtering).
- For the symbol **;** use **&fcmp;** for forward composition of relations and use **&scmp;** for forward composition of schemas.
- I have not figured out how to get fuzz to type check user defined infix functions.
- For the hide operation you cannot use the schema name as an abbreviation; you must give the list of components.

Z mode is identical to math mode with respect to the Weave operation, which produces the Postscript display format.

4 On Entities

Entities are names for special symbols that do not occur in the type font or they must be included without their normal interpretation. For example the symbol < is used to denote the start of an HTML tag so you can not simply write <, instead it is necessary to use its name, <. All FlexOr entities have the standard HTML entity name structure of &entity_name;. If the characters following

the symbol < do not form a valid tag name, then you may type the symbol < directly. Also if the symbol occurs within a quote such as , "<" or '<', then it does not need to be named.

5 Summary of FlexOr tags

The following tags have the described effect. To make it easier to type FlexOr frames with lines that do not wrap, a newline character that immediately follows a tag is not printed, which in some cases is not what is desired. If the newline is needed then insert a space between the tag and the newline, or use a double newline.

Tag	Description
<!−−	The end "tag", "−−>", is required. Enter and leave a meta comment. The tags and the text between the start and end tags are not processed or output by any FlexOr processor. It is a convenient method of temporarily removing text from view. Meta comments can be nested. The outer most comment hides the inner comments.
<ADAPT>	Adapt and include another FlexOr frame. Macros in an adapted frame are used, by the Tangle process, only in the adapted frame. File attributes are **file=filename** and **path=pathname**. The last named path for the adapt tag remains in effect until changed by the occurrence of another path attribute−value pair in an adapt tag. The end tag is required.
<BODY>	The body of the FlexOr frame. The end tag is required.
<B>	Use in documentation mode. Changes font to **bold**. The end tag is required.
 	Inserts a new line for every occurrence. Text begins at the current left margin. Has no end tag.
<CENTER>	Text is centered between the left and right margins. The end tag is required.
<COPY>	Include the contents of a FlexOr frame at tangle time. It is used to physically split large frames into smaller frames, in particular, to be able to reuse frames in different contexts without change (the adapt tag provides for change). Macros in a copied frame are used, by the Tangle process, in the parent frame from the point of the copy. File attributes are **file=filename** and **path =pathname**. The last named path for the copy tag remains in effect until changed by the occurrence of another path attribute−value pair in a copy tag. Has no end tag.
<FLEXOR>	Defines a flexor frame and to set attributes for the weave program. Attributes are **levelmap =n**, **maxline=nnn**, **slideOutput=yes/no**, **secNum=nn**, **tocLevel=n**, **tocPage=yes/no**, **sectionIndex=yes/no** and **variableIndex=yes/no**. The end tag is required.
<I>	Use in documentation mode. Changes font to *italic*. The end tag is required.
<IMG>	Include, at weave time, a diagram encoded in encapsulated Postscript (an eps file). Attributes to change the size and location of the diagram are **xShift=±nn**, **yShift=±nn**, **xScale=nn** and **yScale=nn**. File attributes are **file=filename** and **path=pathname**. The last named path for the image tag remains in effect until changed by the occurrence of another path attribute−value pair in an image tag. Has no end tag. .
<INCLUDE>	Include, at weave time, the contents of a file. Use to include Postscript files such as the header files. The text is included exactly as it without any processing, unlike the copy and adapt tags which process the "included" text. Attributes are **file=filename** and **path=pathname**. The

last named path for the include tag remains in effect until changed by the occurrence of another path attribute−value pair in an include tag. Has no end tag.

<LI> Indicate the start of another list element. See OL and UL tags. The end tag is optional.

<INSERT> Use in the context of adapting a FlexOr frame to insert a new section into an adapted FlexOr frame. The attribute **section=sectionName** is the name of the section to insert. The end tag is optional.

<M> Enter and leave math mode. Use this mode when mathematical notation is written within paragraphs. The end tag is optional.

<MB> Enter and leave math block mode. Use this mode when blocks of mathematical text is written between paragraphs. THe end tag is required.

<MACRO> Define a macro. Macros, used by the tangle process, make it easier to write program text. The end tag is optional.

<OL> Ordered list. List items, indicated with the LI tag, are numbered from 1 up. The end tag is reqired.

<OUTPUT> Change the output file for Tangle. This tag permits the contents of a frame go to different output files. Useful when different versions of a program are the same frame, or when a program consists of parts written in different languages, which are processed by different compilers. Attributes are **file=filename** and **path=pathname**. The last named path for the output tag remains in effect until changed by the occurrence of another path attribute−value pair in an output tag. Has no end tag.

<P> Designates the start of a paragarph. Inserts a new line for every occurrence. Text begins at the current left margin (margin changes with ordered and bulleted lists) plus a paragraph indent amount. Text is filled from succeeding lines. New lines are inserted only when the weave program detects the right margin is being approached. The end tag is optional.

<PRE> Preformatted text. Similar to Program text except that the text is not processed as a program fragment. Font is changed to program font (usually Courier). Text, including new lines is output exactly as it appears in the input. This can be used for simple tables. For programs it is best to use the PT tag. The end tag is required.

<PREFIX> Use in the context of an adaptation to prefix a section in an adapted FlexOr frame. The attribute **section=sectionName** is the name of the section to prefix. The end tag is optional.

<PS> Use for Postscript mode to output Postscript commands to fine tune the output or handle special cases that weave is not programmed to do. Text is output as is without interpretation. The end tag is optional.

<PT> Program text mode. Use this mode to write programs. Tangle knows about this mode and outputs the text to a program file. The Weave process uses a monospace font such as Courier and the text is indented. Newlines are output as they are encountered. The only attribute is **language=[C, C++, Java, Eiffel, None]**. The default value is Java. The end tag is required.

<REF> Reference a section. Use in literate programming in program mode to reference another section. It is similar to a parameterless macro but automatically creates cross−references in the weaved output. Can also be used to reference sections in documentation mode but extra work is needed to adjust the spacing and fonts, as Weave assumes the reference tag is used in program mode. The end

tag is required.

<REPLACE> Use in the context of an adaptation to replace a section in an adapted FlexOr frame. The attribute **section=sectionName** is the name of the section to replace. The end tag is optional.

<SECTION> Denotes a section of a FlexOr frame. Attributes are `level` and `prefix`. The end is required, but occurs at the end of the section name that is given at the beginning of each section. The end of a section is detected by the start of another section or the end body tag.

<SLn> Slide header tag. Use only to make overhead slides but that is now obsolescent by using a laptop for display. The n is the level number where level=0 is the slide title. The end tag is required.

<SUFFIX> Use in the context of an adaptation to suffix a section of an adapted FlexOr frame. The attribute **section=sectionName** is the name of the section to suffix. The end tag is optional.

<TT> Output in a fixed width font. Use to insert snippets of program text into a sentence. For blocks of program text you should use the program text tag. The end tag is required.

<UL> Unordered bulleted list. A bullet is used at all list levels at the start of each list item. The end tag is required.

<V> Verbatim mode. Use to bracket sections of FlexOr frames to be emitted without interpreting the tags for documentation and tutorials. The end tag is required.

<Z> Z mode. Weaves like Math block mode. Use to indicate items that should be output for the Z type checker (Z check button). The end tag is required.

5.1 Tag attribute-value pairs

This section describes the attribute−value pairs that can be used within both start and end tags. See each tag for the attributes that are meaningful for the tag. Multiple attribute−value pairs are written in a space or newline separated list that is terminated by the tag end delimiter '>'.

The value must written within double quotes if it contains a space, otherwise the use of double quotes is optional. For example, you can write either file=mainName.eps or file="mainName.eps" but you must write format="−5 vtab", since the space between the 5 and vtab is a part of the value.

- **color=colorName** – Use to change the color of the current font, where predefined colour names are white, black, lightgray, darkgray, blue, brown, cyan, green, greendark, magenta, yellow; for example, `color=green`. The attribute is used most frequently with the tags <B>,
,<M>,<P> and <SL>.
- **file=filename** – Use in tags that read from or write to files.
- **format="...Postscript commands..."** – The most common use is to change the vertical spacing between components. For example `format=newpage` and `format="-5 vtab"`. For a start tag the format takes place before the default formatting for the tag. For an end tag the format takes place after the default formatting for the tag.
- **language=[C, C++, Java, Eiffel, None]** – If the language attribute is used in a PT tag, then keywords for C, C++, Java or Eiffel can be differentiated from variables so keywords are not put into the variable index. The default language is Java, as a consequence Java keywords are excluded from the variable index, which will exclude valid variable names for programs in other languages, thus the value "None" is available to indicate that keywords are not to be looked for and all variables will be put into the variable index. Use in the PT tag.
- **leaderStyle** – If the Leaderstyle value is yes, then the table of contents appears as

follows
sectionName leader section−level
whereas, a value of no causes the table of contents to be the style
section−level sectionName
Note that the leaderStyle attribute must be defined before the tocLevel attribute in a FlexOr tag.

- **level=n** – Use to indicate the level of a section within a document. Level 0 is the highest level, level=1 is the next lowest, etc. The attribute is only meaningful in a `SECTION` tag. The level is retained until explictily changed, so you do not have use the attribute until a change in level is needed.
- **levelmap=n** – Use to map section level 0 to a different typesetting level. For example, `levelmap=2` means that section level=0 is mapped to head font 2, and section `level=1` is mapped to head font 3, etc. It is only meaningful in the `FLEXOR` tag.
- **maxline=nnn** – Use to set the maximum line length for fill operations. For example, for the the text font size defined in `FlexOr.header` a line length of approximately 100 characters fits between the left and right margins, so `maxline=100` is used. For slides, however the font size is larger thus fewer characters fit on a line, so `maxline=50` may be appropriate. This attribute decouples the defintions of font size in the Postscript header files and the algorithm used by the Weave operation.
- **path=pathname** – Use to define the path from the root to the filename for input and output tags; for example, path=`/cs/dept/www/java/bin/FlexOr/frames`. The pathname is retained until it is explicitly changed, so you do not have to use the attribute until a change in pathname is needed.
- **prefix="... a string ..."** – Define a prefix for the succeeding section names. Useful, for example, for appendices where you may want to have every section beginning with "A−1 " in the weaved output. The value remains until redefined. Default is the empty string. Only meaningful for the `SECTION` tag.
- **sectionIndex=yes/no** – Use yes to have a section index in the weaved output. Only valid for the FLEXOR tag.
- **secNum=nn** – Use to set the number to use at level zero of a section number. The attribute is only meaningful in the `SECTION` tag. Use in the first section of a FlexOr frame that is a part of a larger logical document. For example, a large logical frame may be partitioned into a set of physical frames, where each physical frame corresponds to a different section in the logical frame. In such a case, each physical frame needs to encode its section number using this attribute.
- **slideOutput=yes/no** – Set to yes if output is to produce overhead slides. Default is no. Only meaningful for the `FLEXOR` tag. This attribute is obsolescent and may not work correctly with the latest changes made to FlexOr. Slides have been replaced by direct display from a laptop.
- **tocLevel=n** s– Sets the maximum section level to include in the table of contents. Using the attribute causes the table of contents to be output after the Prolog and before the first section. If the attribute is not used, then no table of contents is output. The attribute is only meaningful in a FLEXOR tag.
- **variableIndex=yes/no** – Use yes to have a variable index in the weaved output. For the languages C, C++, Eiffel and Java the index excludes keywords if you use the language attribute in the first PT tag in the frame. Only valid for the FLEXOR tag.
- **xShift=±points**, **yShift=±points**, **xScale=factor**, **yScale=factor** – Use in the `IMG` tag when including eps files (Encapsulated Postscript). Use xScale and yScale to scale the size of the included diagram, where 100, the default value does not change the size of the object. Larger values increase the size of the object, thus `yscale=150` increases the size by 50% in the y direction. Use xShift and yShift to shift the diagram with respect to the origin by the number of specified points; positive values are to the right (x) and down (y),

negative values to the left (x) and up (y).

Due to the properties of the current algorithm, you have to use shift whenever scale is used, as scaling causes the left and bottom edges of the diagram to extend to the left and the bottom. Currently, the shift amount is found by trial and error, and experience. The following is an example.

<IMG xshift=−130 yshift=−100 xscale=175 yscale=175 file=cal−7.eps>

6 List of some useful Postscript commands

One can view the various header files and find the available commands. They have the following structure

```
/commandName { definition } def
```

You can recognize them with the /... def combination. Most of the commands are documented with comments (begin with % and extend to the end of the line).

Postscript is a stack interpreter, which means that all the operands for a command are on the stack (in the order expected by the command). The command takes its operands from the stack, does its work and leaves its results (if any) on the stack for the next command.

Strings are written within (), as in (This is a string). Backslash ,\, is the escape character – need to escape \, (and); for example the string "(/)" in Postscript is written as (\(\\ \)).

Command	Description
show	String is on the stack, display the string on the page at the current point
nl	Go the beginning of the next line in the document
hr	Draw a horizontal rule from the current point for a length that is on the stack. There is a fixed default width of the rule line.
vtab	Move up or down by a vertical tab distance in points that is on the stack. Positive is down the page, negative is up the page.
tab	Move left or right a horizontal tab distance in points that is on the stack. Distance moved is from the last defined value of hpos (horizontal position). Positive input moves to the right, negative input moves to the left.
atab	Move left or right a horizontal tab distance in points that is on the stack. Distance moved is from the left margin. Positive input moves to the right, negative input moves to the left.
centerText	String on the stack is centered on the current line between the left and right margins.
rightJustifyText	String on stack is right justified on the current line with respect to the right margin.
newpage	Show the current page and start a new page.

Program text is not referenced

7 Including encapsulated Postscript diagrams

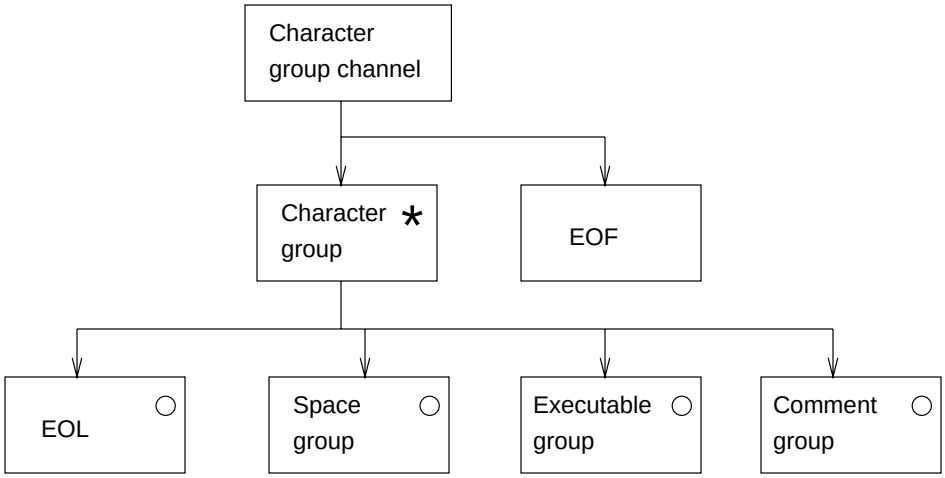
Instructions for including an Encapsulated Postscript diagrams in FlexOr frames. Each diagram must be a separate file.

In your FlexOr frame insert the following command to include the diagram where you want it.

```
<IMG path="thePath/" file=the_file_name.eps>
```

For the following diagram, I used the following command sequence.

```
<PS>10 vtab </PS>
<IMG path="/cs/dept/www/java/bin/FlexOr/examples"
     file=InstrDiagInFlexOr.xfig.eps>
```



As you can see the diagram appears as expected.

If multiple images are included in sequence from the same directory, then the path only needs to be specified for the first image, as that path will be used by default for succeeding images until the path is explicitly changed.

Attributes to change the size and location of the image are **xShift=±nn**, **yShift=±nn**, **xScale=nn** and **yScale=nn**.

Program text is not referenced

8 Macro definitions

A modest macro definition facility is available in FlexOr to simplify writing program text. The macro facility does not have choice or looping constructs. However macros can be used within macro definitions as long as there is no loop in calls, including no recursion.

A simplified grammar definition of a macro is the following. Spaces and newlines can be in the macro header but are not a part of the definition and are skipped over in matching the header to a call sequence. Spaces and newlines are a part of the program text in the macro body.

```
Macro ::= MacroHeader = MacroBody ;
MacroHeader ::= MacroName +[ NoiseWords , Parameters ] ;
MacroBody ::= +(ProgramText , Parameters) ;

MacroName ::= ProgramIdentifier ;
```

```
NoiseWords ::= any program text, excluding spaces, newlines and parameters ;
Parameter ::= '#' ProgramIdentifier
```

Macros can be used with no parameters to name constants, as in the following, which defines the length of lines for section names printed in various contexts.

```
<MACRO format="6 vtab">LEN_ADAPT_CMD_LINE = 80</MACRO>
<MACRO>LEN_SECTION_LINE = 40</MACRO>
<MACRO>LEN_REFERENCE_LINE = 80</MACRO>
```

weaves to look like the following.

```
DEFINE LEN_ADAPT_CMD_LINE ::= 80
```

```
DEFINE LEN_SECTION_LINE ::= 40
```

```
DEFINE LEN_REFERENCE_LINE ::= 80
```

The following example shows how noise words can be used to make a macro more readable. In this case the program text that decides if the current component contents should be put on the current line or at the beginning of a new line.

```
<MACRO format="6 vtab">output_contents on current or next line =
if (lineLength + component.theContents.length() > MaxLineLn) {
    outputOutputStr(SHOW_NEWLINE); }
 addToOutputStr(component.theContents);</MACRO>
```

weaves to look like the following.

```
DEFINE output_contents on current or next line ::=
if (lineLength + component.theContents.length() > MaxLineLn) {
      outputOutputStr(SHOW_NEWLINE); }
  addToOutputStr(component.theContents);
```

The following example shows the use parameters. The example is taken from the FlexOr Weave process to construct custom versions of visit routines for the adaptation commands `Insert`, `Prefix`, `Replace`, and `Suffix` have the same semantic actiions. The differences are in the routine name and in the output keyword.

```
<MACRO format="6 vtab">output_adaptation command #cmd using #keyword =
public void visit#cmd(#cmd component) {
 outputOutputStr(NO_NEWLINE); lineLength = 0;
 processAttributes(component);
  output.write("\n" + #keyword);
  sectionNameOutput(component.section, LEN_ADAPT_CMD_LINE, " 18 tab ");
  addToOutputStr(" >>");
  outputOutputStr(SHOW_NEWLINE);
  output.write("ProgFont \n");
 processComponentList(component.theChildren);
 processAttributes(component.theEndTag);
}</MACRO>
```

weaves to look like the following.

```
DEFINE output_adaptation command #cmd using #keyword ::=
public void visit#cmd(#cmd component) {
  outputOutputStr(NO_NEWLINE); lineLength = 0;
  processAttributes(component);
    output.write("\n" + #keyword);
```

```
    sectionNameOutput(component.section, LEN_ADAPT_CMD_LINE, " 18 tab ");
    addToOutputStr(" >>");
    outputOutputStr(SHOW_NEWLINE);
    output.write("ProgFont \n");
  processComponentList(component.theChildren);
  processAttributes(component.theEndTag);
}
```

The following example was used in a tag based version of FlexOr to create the text files for the leaf components.

<MACRO format="6 vtab"> programText #component #additionalText =
package FlexOr.sgmlCB;
public class #component extends LeafComponent {
 public #component(int theCharNumber, int theLineNumber, String theContents) {
  super(theCharNumber, theLineNumber, theContents);
 }
 public void accept(SGMLvisitor visitor) { visitor.visit#component(this); }
#additionalText
}</MACRO>

weaves to look like the following.

```
DEFINE programText #component #additionalText ::=
package FlexOr.sgmlCB;

public class #component extends LeafComponent {

  public #component(int theCharNumber, int theLineNumber, String theContents) {
    super(theCharNumber, theLineNumber, theContents);
  }

  public void accept(SGMLvisitor visitor) { visitor.visit#component(this); }
#additionalText
}
```

The following shows how the macro `programText` can be used. Notice the used of [ ] to define the program text that corresponds to the parameter #additionalText. Any of () [] and {} can be used delimit an arbitrary sequence of program text to correspond with a parameter. The enclosing () [] and {} are not a part of the argument.

```
programText MathKeyword [
  public String argument = null;

  public String toString() {
    if (argument == null) return theContents;
    else return theContents + argument;
  }]
```

The following shows how a null sequence can be passed as an argument to a macro. In this case the parameter #additionalText is the null sequence of characters.

```
programText Word []
```

Program text is not referenced

9 Using mathematical notation

This section describes and is an example of how to embed mathematical symbols into FlexOr frames. Use a Postscript Weave.

9.1 Simple example

All mathematical notation should appear within the <M> and </M> tags. For example the following commands

```
<M>     &all;x,y : &Nat; &cbar; x &lt; y &bul; x&sup2; &lt; y&sup2;</M>
```

create the following mathematical expression. Note the lead spaces to get indentation.

$$\forall x,y : \mathbb{N} \mid x < y \bullet x^2 < y^2$$

This sentence contains the math snippet $\forall x,y : \mathbb{N} \mid x < y \bullet x^2 < y^2$, where the lead spaces are not typed as the spacing is not needed.

All mathematical notation that you want to appear as a block of consecutive lines should appear within the <MB> and </MB> tags, they simplify the formatting before and after the block of math text. The following commands

```
<MB>
double_someIntegers == { i : &Int; &cbar; i = 1 &or; i = 3 &or; i = 15 &or; someIntegers == {1, 3, 15, −5, 0}
double_someIntegers_1 == { i : &Int; &cbar; i &mem; someIntegers &bul; 2 * i }
double_someIntegers_2 == { i : someIntegers &bul; 2 * i } i = −5 &or; i = 0 &bul; 2 * i }
</MB>
```

create the following math block.

$$\begin{aligned}
&\text{double_someIntegers} == \{ i : \mathbb{Z} \mid i = 1 \lor i = 3 \lor i = 15 \lor \text{someIntegers} == \{1, 3, 15, -5, 0\} \\
&\text{double_someIntegers_1} == \{ i : \mathbb{Z} \mid i \in \text{someIntegers} \bullet 2 * i \} \\
&\text{double_someIntegers_2} == \{ i : \text{someIntegers} \bullet 2 * i \\
&i = -5 \lor i = 0 \bullet 2 * i \}
\end{aligned}$$

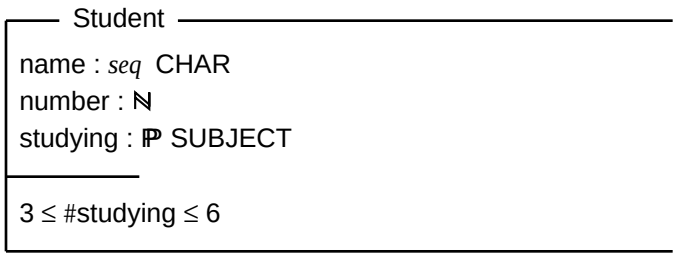
9.2 Z Schemas

FlexOr math mode has math keywords that begin with colon and end with period that process the subparts of the Z constructs. For schemas and axioms you must make sure the entire structure fits on one page or else the vertical line is not created corrected by :eschema. and :eaxdef.

Keyword	Description
:schema.	Start of a schema with the given name
:zwhere.	Separate the data part from the predicate part of a schema or axiom.
:eschema.	Terminate a schema
:generic.	Start a generic schema with the given paramters
:egeneric.	Terminate a generic schema
:axdef.	Start of an axiomatic definition
:eaxdef.	Terminate an axiomatic definition

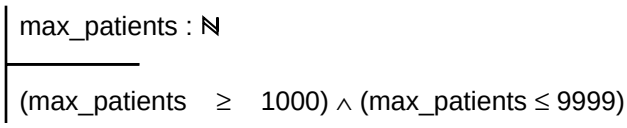
The schema shown below is created with the following commands.

```
<Z>
:schema.Student
name : &seq; CHAR
number : &Nat;
studying : &pset; SUBJECT
:zwhere.
3 &leq; &num;studying &leq; 6
:eschema.
</Z>
```



The axiom definition shown below is created with the following commands.

```
<Z>
:axdef.
max_patients : &Nat;
:zwhere.
(max_patients &geq; 1000) &and; (max_patients &leq; 9999)
:eaxdef.
</Z>
```



The generic defintion shown below is created with the following commands.

```
<Z>
:generic.[FromSet,ToSet]
&dom; : (FromSet &rel; ToSet) &tfun; &pset; FromSet
&ran; : (FromSet &rel; ToSet) &tfun; &pset; ToSet
:zwhere.
&all; R : FromSet &rel; ToSet &bul;
      &dom; R = { from : FromSet; to : ToSet  &cbar;  (from , to) &mem; R  &bul;  from }
    &and;
      &ran; R = { from : FromSet; to : ToSet  &cbar;  (from , to) &mem; R  &bul;  to }
:egeneric.
</Z>
```

$$[FromSet, ToSet]$$

$$dom : (FromSet \leftrightarrow ToSet) \rightarrow \mathbb{P}\, FromSet$$
$$ran : (FromSet \leftrightarrow ToSet) \rightarrow \mathbb{P}\, ToSet$$

$$\forall R : FromSet \leftrightarrow ToSet \bullet$$
$$dom\ R = \{ from : FromSet; to : ToSet \mid (from , to) \in R \bullet from \}$$
$$\wedge$$
$$ran\ R = \{ from : FromSet; to : ToSet \mid (from , to) \in R \bullet to \}$$

9.3 Mathematical symbols and SGML notation

9.4 Set notation

equal	=	&eq;	proper subset	⊂	&psubs;
member	∈	&mem;	not member	∉	&nem;
power set	ℙ	&pset;	distributed intersection	⋂	&dint;
product	*	∏	distributed union	⋃	&duni;
not equal	≠	&neq;	finite set	𝔽	&fset;
intersection	∩	∫	finite set, not empty	$\mathbb{F}_1$	&fset1;
union	∪	&uni;	power set, not empty	$\mathbb{P}_1$	&pset1;
difference	\	&diff;	null set	∅	&null;
subset	⊆	&subs;	bar	|	&cbar;
superset	⊇	&sups;	proper superset	⊃	&psups;
left set	{	&lset;	right set	}	&rset;

9.5 Sequence and Bag notation

sequence	*seq*	&seq;	distributed override	⊕/	&dovr;
non empty sequence	seq_1	&seq1;	distributed composition	;/	&dcmp;
items	*items*	&items;	distributed concatenation	^/	&dcat;
count	*count*	&count;	index restriction	↾	&ires;
first	*first*	&first;	rear	*rear*	&rear;
front	*front*	&front;	sequence restriction	↿	&sres;
last	*last*	&last;	bag	*bag*	&bag;
head	*head*	&head;	in bag	*in*	&inbag;
tail	*tail*	&tail;	union plus	⊎	⊎
reverse	*rev*	&rev;	disjoint	*disjoint*	&disjoint;
squash	*squash*	&squash;	partition	*partition*	&partition;
concatenation	^	&cat;	injective sequence	*iseq*	&iseq;
left sequence	⟨	&lseq;	right sequence	⟩	&rseq;
left bag	⟦	&lbag;	right bag	⟧	&rbag;
prefix of	≤	≤	symbol count	↓	&arrd;
s repeated n times	s^n	s:ups. n	like - n symbols removed	$\overset{n}{\leq}$	:like. n
nil sequence	⟨⟩	&nseq;			

9.6 Relational notation

relation	↔	&rel;	range anti-restriction	⩥	&rsub;
identity	*id*	&id;	function override	⊕	&fovr;
domain	*dom*	&dom;	left image	⦇	&limg;
range	*ran*	&ran;	right image	⦈	&rimg;
forward composition	;	&fcmp;	inverse	$^{-1}$	&inv;
composition	∘	&cmp;	inverse tie	~	&invti;
domain restriction	◁	&dres;	reflexive transitive closure	*	&rtcl;
domain subtraction	⩤	&dsub;	transitive closure	$^{+}$	&tcl;
range restriction	▷	&rres;	map	↦	↦
iterate	*iter*	&iter;	first	*first*	&first;
second	*second*	&second;			

9.7 Function notation

partial function	⇸	&pfun;	finite injection	⤔	&finj;
total function	→	&tfun;	partial surjection	⤀	&psur;
finite function	⇻	&ffun;	total surjection	↠	&tsur;
partial injection	⤔	&pinj;	bijection	⤖	&bij;
total injection	↣	&tinj;			

9.8 Natural numbers

natural numbers	ℕ	&Nat;	division	÷	÷
natural numbers, >0	$\mathbb{N}_1$	&Nat1;	modulus	*mod*	&mod;
integers	ℤ	∬	reals	ℝ	&Real;
successor	*succ*	≻	greater than or equal to	≥	≥
predecessor	*pred*	&pred;	less than or equal to	≤	≤
minimum	*min*	&min;	greater than	>	>
maximum	*max*	&max;	less than	<	<
times	×	×	minus	−	−
number of	#	#	plus	+	+

9.9 Logic notation

for all	∀	&all;	there exists	∃	&exi;
there exists one	$\exists_1$	&exi1;	not	¬	¬
true	*true*	&true;	false	*false*	&false;
and	∧	∧	or	∨	∨
equal	=	&eq;	not equal	≠	&neq;
implies	⇒	&imp;	if and only if	⇔	⇔
bullet	•	&bul;	end proof	□	&qed;

9.10 Schema notation

Delta	Δ	Δ	Xi	Ξ	Ξ
pre	*pre*	⪯	zfor	/	&zfor;
schema definition	$\hat{=}$	&sdef;	for all	∀	&all;
there exists	∃	&exi;	there exists one	$\exists_1$	&exi1;
not	¬	¬	and	∧	∧
or	∨	∨	implies	⇒	&imp;
if and only if	⇔	⇔	projection	↾	&sproj;
hide	\	&hide;	compose	⨾	&scmp;
override	⊕	&zovr;	pipe	≫	&zpipe;

9.11 Guarded command language notation

refer by	≼	&refby;	skip	*skip*	&skip;
abort	*abort*	&abort;	assignment	:=	&ass;
if	*if*	&gif;	end if	*fi*	&gfi;
gbar	|	&gbar;	guards	→	&guards;
do	*do*	&gdo;	end do	*od*	&god;
wdef	$\stackrel{\tau}{=}$	&wdef;	choice	□	&choice;

9.12 CSP special events notation

success	✓	&success;	catastrophe	↯	&catas;
exchange	⊗	&xch;	checkpoint	©	&chkpt;
acquire	*acquire*	&acquire;	release	*release*	&release;

9.13 CSP processes notation

alphabet of	α	α	then	→	&then;
choice bar	|	&cbar;	mu	μ	μ
after	/	&after;	in parallel with	‖	&parwth;
or	⊓	&cspor;	choice	□	&choice;
without	\	&wthout;	interleave	⫴	&intlve;
chained to	⪢	&chnto;	subordinate	//	&subord;
but on catastrophe	$\hat{\downarrow}$	&butcat;	restartable P	$\hat{P}$	P&restart;
P alternating Q	P⊗Q	P &xch; Q	repeat P	*P	&repeat;P
if	◁	&cspif;	else	▷	&cspelse;
satisfies	**sat**	&sat;	traces	*traces*	&tr;
accessible(P)	*acc* (P)	&acc;(P)	as good as, as much as	⊆	&subs;
e is defined	*D* e	&def;e	final value of x	x’	x’
interrupt	▽	&intrpt;	interrupt on catastrophe	⩛	&intcat;
refusals	*refusals*	&ref;	assignable variables	*var* (P)	&var;(P)
diverges	*diverges*	&divrgs;	divergences	*divergences*	&divgen;
interleaves	*interleaves*	&intlvs;	chaos	⊥	&chaos;
failures	*failures*	&fail;			

9.14 Miscellaneous symbols

left square bracket	[	[	right square bracket	]	]
lambda	λ	λ	mu	μ	μ
subscript 0-9	$_{0}$ – $_{9}$	&sub0;-&sub9;	superscript 0-9	^{0} – ^{9}	&sup0;-&sup9;
turn	⊢	&turn;	abbreviation	==	&tdef;
datatype definition	::=	&ddef;	theta	Θ	θ
left angle	⪡	⟨	right angle	⪢	⟩
left parenthesis	(	&lp;	right parenthesis	)	&rp;
left arrow	←	&arrl;	right arrow	→	&arrr;
up arrow	↑	&arru;	down arrow	↓	&arrd;
double left arrow	⇐	&darrl;	double right arrow	⇒	&darrr;
double up arrow	⇑	&darru;	double down arrow	⇓	&darrd;
both way arrow	↔	&arrb;	double both way arrow	⇔	&darrb;
angle	∠	∠	gradient	∇	&grade;
registered	®	&regter;	copyright	©	&cpyrt;
equivalent	≡	&equi;	congruent	≅	&cgrnt;
summation	Σ	σ	perpendicular	⊥	&pdic;
underscore	_	&unscr;	Pi	π	π
similar	~	∼	minute	*min*	&min;
second	″	&sec;	degree	°	&degree;
infinity	∞	&inf;	florin	*f*	&florin;
plus minus	±	&plumin;	proportional	∝	∝
partial differential	∂	&pdiff;	approximation	≈	&apequ;
ellipsis	…	ℓ	product	∏	&dprod;
radical	√	&rad;	dot in math	·	˙
lozenge	◊	◊	such	ɜ	&such;
then	*then*	&pthen;	else	*else*	&pelse;
colon	:	:	multiply	*	&mul;

9.15 Miscellaneous combination symbols

Raise +	A^{+}	A&rplus;	Raise *	A^{*}	A&rast;
superscript a string	ab^{de}	ab:ups. de	subscript a string	ab_{de}	ab:dns. de
str over prev symbol	$\overset{de}{ab}$	ab:ovrs. de	string under prev symbol	$\underset{de}{ab}$	ab:unds. de
timeout interrupt	$\overset{time}{\triangledown}$	:timeout. time			