The *xlincks* User's Manual for
Version 2.2 of the **LINCKS** Database System
Manual version: 1.16 5/27/94

Lab for Intelligent Information Systems
Department of Computer and Information Science
University of Linköping, Sweden

Editor: David Partain

Contributors: Ralph Rönnquist, Lin Padgham, Martin Sjölin.

Last updated: 5/27/94

Abstract

This manual for version 2 of the **LINCKS** system is designed to, as simply as possible, describe the use of the *xlincks* interface to the **LINCKS** system. We have attempted to cover all of the functionality of the current system by providing both task-oriented and command-oriented documentation.

E-mail: lincks@ida.liu.se
Bug Reports: lincks-bugs@ida.liu.se

Contents

Chapter 1

Introduction to the System

1.1 Roadmap for this manual

Do you need to use this manual? We hope that you will only need to use it for a short time until you are familiar with the system. Thereafter, we hope that the on-line help will be sufficient to guide you. However, when that is not enough, this manual should provide you with what you need.

This document is for those who plan to use the *xlincks* interface to the **LINCKS** database system. It will guide you in performing both the basic and more advanced tasks. We will do this by:

- giving a brief introduction to the **LINCKS** model and *xlincks* as well as some of the common vocabulary used throughout,
- going through some of the paradigms used by the interface,
- going through in some detail in chapters 2 and 3 the tasks that you may want to do within *xlincks*,
- giving an individual description of each possible command in chapter 4,
- trying to answer some of the more frequently asked questions in chapter 5.
- describe how to export (dump) parts of a database as a t2lincks file in chapter 6.
- a brief documentation on how to write general presentation descriptor (GPD) in chapter 7.
- defining some commonly-used terms in chapter 8.

1.2 Overview of LINCKS

The **LINCKS** system is a prototype multi-user database system which has been developed as a base for research in advanced information systems and for applications which require both artificial intelligence techniques and database aspects. **LINCKS** can be seen as an object-centred database management system for applications which require more flexible data representations than are manageable in traditional relational databases.

The database is made up of small semi-structured *information component*. One part of each information component consists of attributes and their values. This is structured information (reminiscent of a frame-like representation) which can easily be used by software programs for various kinds of decision making and support. The second part of each information component is unstructured information which is expected to be meaningful to the user, but which we do not expect computer software to analyse - e.g. free text, bitmaps, audio information, etc. We call this the *image*. The third part of each information component is the links. These are used to connect information component together, either to form composite objects, or simply to establish a connection or relation. Figure 1.1 shows such a information component.

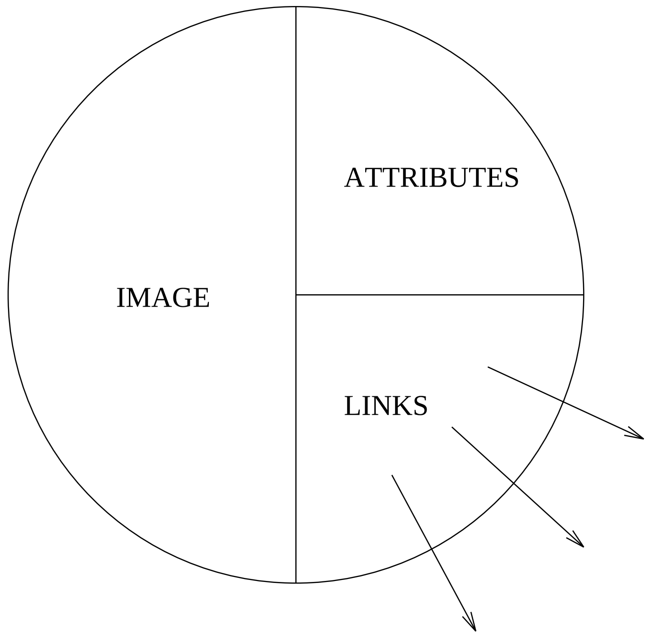

Figure 1.1: Parts of an 'information component'

LINCKS supports much of the functionality which also exists in other systems, such as an object-centred model, hypertext linking and browsing, and flexible views. The two things which **LINCKS** gives support for beyond that found in other systems are support for composite shared objects and support for maintenance of more extensive history information than is usual.

1.2.1 Composite Objects and Views

A user of the system typically works with *composite objects* which are built up out of these small information components (generally referred to simply as 'components' when used in reference to a part of a composite object). For instance, a document may be built up out of components representing main sections, where each main section is built up of subsections which are in turn built up of paragraphs. Each composite object can thus be seen as a tree structure, which is built from information components in the underlying database. A single information component can be a component in many different composite objects - perhaps used and developed by different people. Thus information is shared and reused.

When a user works with a composite object s/he does this via a *view* of that object. A view picks out selected parts from the components and combines them in a single display window. A view is made up of *items* where an item is the smallest indivisible part of a component - e.g. an attribute value, a bitmap or a link tag.[1] A view may contain none, one or many items from each component. For example one view of a document (the most familiar one) could be a display where the title and the authors are extracted as attribute values from the root component, an abstract is extracted from the image of an abstract component, section and subsection titles from attribute values of components, and the paragraphs from the images of paragraph components. These are then displayed in the appropriate order in the display window. An alternative view may be one that selects title and authors, an abstract, section and subsection titles, but instead of paragraphs displays only a summary of each subsection taken from the image of that component.

A composite object is in fact defined by the combination of a *view descriptor* (which we call a Generic Presentation Descriptor) and a particular root component. The view descriptor gives a template for the composite object type (such as 'full document', 'memo', 'folder', etc.), while the root component indicates the particular place in the database from which the components for this particular composite object can be found.

1.2.2 History

The **LINCKS** system maintains a substantial amount of historical information. The motivation for this is partly that intelligent support or A.I. reasoning often requires information not only about the present state of affairs but also about past states. An additional motivation is that even apart from automated reasoning, users often wish to have access to historical information.

The **LINCKS** system maintains three main types of history information:

- Temporal history, or logical transaction time history.

  The temporal history in **LINCKS** could be described as a history based on logical transaction time. That is, it is based on transaction time but at a logical level rather than at the level of time stamps regarding when objects are saved. Thus if *user 1* takes out an object, and edits it, then *user 2* takes out the same object and edits it (prior to *user 1* having done a save), then *user 1* saves, and finally *user 2* saves, then the two edits are regarded as being parallel in the temporal history. *User 1*'s edit is not before that of *user 2*.

  The temporal history is the only history that is currently shown by the system. It can be shown at the level of the composite object or at the level of individual components. One can mark and view previous versions and can also promote them to be current in order to continue development from that point.

- Development history.

[1] We currently are unable to include non-textual items in our views. The underlying database is, however, able to store non-textual information, and we plan to incorporate this into the display layer in the near future.

Development history represents the information regarding where an object has come from in its development. It may often be natural to have an object in several versions, which develop in parallel, at least for a period of time. Two users may wish to develop their own versions of a document which they are working on together, rather than trying to combine them at every step.

Development history is constrained by the temporal history in the sense that a version cannot be a predecessor in a development history unless it is also a predecessor in the temporal history (or from a separate object[2]).

There is currently no graphical interface to the development history. It is carried in the **SYSTEM:Parent** link which can be seen in the node view.

- Command history.

  The command history gives information regarding the (logical) temporal relation between commands that have been issued by various users of the system. It maintains the temporal relationships between database commands (store and retrieve) issued by different users - thus one can ask questions of the type, "Was a given object retrieved by user 1 before another object was retrieved by user 2?" This can be useful for providing assistance to users in reasoning about why things have happened as they have.

  The command history also contains information about what objects have been affected by each command. Thus one can ask the question, "What command (including person and time) caused this particular change in the data?"

  There is currently no graphical interface to the command history, but we plan to include one soon. This is an area of active research for us, and we believe that the command history gives possibilities for more advanced reasoning about what has happened in a database than has previously been usual.

Bound and Unbound References

The objects in the system (both composite objects and individual information components) are actually structures of versions of the object as they have existed at various times. A reference to an object can thus be either a reference to a particular version (a bound reference) or an open reference (unbound) which is resolved at the point that the reference is used. It is resolved currently to be the latest version in the temporal history although in later versions we may have a more complex resolution which also uses the development history. The use of unbound or open references ensures that one always gets the latest version of a component in a composite object. That is, if you have an unbound reference to a 'person object' containing information about a person, you'll always get the latest information even as the components (telephone number, address, office, employer) change.

The use of bound references ensures that one will always get a particular version of the object, regardless of whether it changes in the future. This can be useful for representing such things as printouts, where one wishes to match against an external existence.

[2]We allow a development history for an object to incorporate references to other objects, whereas the temporal history is only within a (possibly composite) object.

1.2.3 The Software Layers

The **LINCKS** system is made up of essentially four separate layers. Going from the layer closest to the data storage, to the layer closest to the user, the layers are the following:

1. The NODE database layer
2. The NODE workspace layer
3. The Composition Management layer
4. The interface layer

The basic architecture is a client server architecture, where the database resides on a single server machine[3] and the clients are on workstations connected to the server by a local area network.

- Layer 1 (The NODE database layer) resides on the server machine and is made up of three modules (monitor, dbs, and netserv) which manage such things as attaching a workstation process to a database, low-level retrieval and storage, and awareness of parallel editing.
- Layer 2 (The NODE workspace layer) is encapsulated in the LIBLINCKS library and works together with layer 1 to form the basic object-centred database management system which we refer to as NODE [4]. NODE has previously been released separately to a number of users. The NODE interface is a programmer's interface only and is via the "user functions" in the LIBLINCKS library.

  The central data structure of the NODE subsystem is the information component described earlier.
- Layer 3 filters out substructures in the database, so called composite objects, and represents these as a finer grained structure of items. The central data structure of this layer is what we call the *reference structure*. The reference structure consists of a template which gives the form of a particular type of composite object, and the target which gives the form of a particular instance of the template.
- Layer 4 displays the leaf items of the *reference structure* in a window display. It provides the interface to the editing of the reference structure and via that, the information components. Editing of the leaf items is done directly on the display using *emacs*-like commands. Editing of the inner structure requires structure editing commands, which are outside the scope of editors like *emacs*.

  Our goal is for layer 4 to be replaceable by modules relying on different windowing systems (or even text terminals). However the current version requires X Windows.

[3] The aim is to allow multiple database servers with varying types of databases, but this is not realised in the current version of the system.

[4] Lin Padgham, **NODE**: A Database for Use by Intelligent Systems, Proceedings of Third International Symposium on Methodologies for Intelligent Systems, October 12–15, 1988

1.2.4 Potential Uses of LINCKS

Information System Applications

Many information system applications are centred around different kinds of forms and other support for entering data, and different ways of viewing and browsing the data in the system. These kinds of applications are the most straightforward for the **LINCKS** system.

There are two basic design tasks in building such an application in **LINCKS**. The first is to determine how the application data should be organised in the database (with particular consideration given to sharing of data). The second is to determine the various views and forms that will be needed for browsing existing data and entering new data.

Once these tasks are done the application builder simply creates *Generic Presentation Descriptors (GPD's)* in accordance with these design decisions and enters the application data. Information on how to create GPD's can be found in chapter 7 of the manual. If there is a lot of data to be entered which already exists in electronic form it is often easiest to write a small program to enter this data into the database. It may also be possible to use the program *t2lincks* for this purpose.

Applications with Added Functionality

The base system provides functionality for editing and browsing in the database in an advanced manner. If, however, the application requires specific functionality outside of that already provided, then this requires some additional work. Extra functionality which does not require additional kinds of interaction than that already provided is relatively easily added. Menu choices indicating the additional functionality are added to an existing menu (or displayed on a separate menu), allowing an entrance to the new functionality. Following the computations of the new function, the results are displayed by having the application create or modify an information component which can then be displayed in the normal manner to view the results.

An example of such functionality would be classification. Objects can be created and defined according to a form given by a GPD. A classification menu choice can then allow a chosen object to be classified into a taxonomy, with new links and new information being created as necessary.

Another example of such added functionality (in a hospital information system) would be to calculate the number of patients currently admitted in a particular ward. The menu choice would be added allowing the user to select this option and pass the required parameter (the ward object). The application routine would perform the necessary calculations and store the result in a new or existing DB object. The function may then request display of the resulting DB object if it is not already visible.

Applications with Specialized Interface Functionality

If the added functionality requires a new kind of interaction, for example specialized graphics, then the addition is more complex. A new kind of display widget must be built (using in our case the X Windows system) supporting the desired interface functionality. A module must also be added to build the new type of widget from the reference structure in the composition management layer. Finally, a way must be devised by which a particular item in the view can be distinguished and thereby displayed in the new widget.

1.3 Paradigms used in *xlincks*

1.3.1 Editing

Part of the basic *xlincks* model is that one is always in an editor. Everything that is visible can also be edited directly, without any explicit mode shift.

A user always works with a composite object via some view of that object. The aim in *xlincks* is to allow the user to work with composite objects as whole entities rather than requiring a conscious awareness of the individual components. The style of editing in a composite object is based on the text editor *emacs*. Many of the commands are the same as in *emacs* and editing ranges over the entire composite object, just as if it was a single file being edited.

As much as possible, familiar *emacs* commands perform the same way in *xlincks* as they do in *emacs*. For example, commands such as *ctrl*-a take you to the beginning of the line, *ctrl*-e goes to the end, *ctrl*-d deletes a character, and so forth. A complete list of these commands is given in section 4.3.

Some additional editing capabilities arise because of the underlying structure of the representation (e.g. making a new section requires a new type of command as it has no equivalent in *emacs*). Also, ordinary *emacs* operations such as deleting a part of a document may require reorganisation of the underlying structure of the database. We refer to these two classes of editing operations as *structure editing* All of these **LINCKS**-specific commands are prefixed by *meta*-l (for **LINCKS**). For example, the command to show the history of a composite object is *meta*-l *meta*-h. Again, a list of these commands is given in section 4.3.

The *xlincks* approach to structured editing is to give as much automated support as possible to the user in maintaining and creating the underlying database structure, without requiring that s/he explicitly indicate what database structure is desired. Naturally, there are some cases where the user needs to understand how the system makes decisions in order to be able to carry out the desired operation. However, our intention is that in cases where intentions should be clear (for example, to another person) the system will react as desired without the user having any deeper understanding of the system.

1.3.2 The X-based user interface

For those familiar with the X Window System or some variant thereof, the X interface used in *xlincks* should be relatively simple to understand. There are two ways to interface with the *xlincks* functionality: (1) through buttons on the Command Menu and in each object window, and (2) using keyboard commands, which often begin with *meta*-l.

On all of the buttons, clicking on the button with the left mouse button will initiate the operation. Clicking with the right button will pop up a help menu about that button. Clicking with the middle button does nothing.

There are two sets of buttons you will use. The first are the buttons in Command Menu (see chapter 8). These buttons operate on a single item (i.e. 'Item History'), on all windows (such as 'Store All'), on neither items or windows (i.e. 'User Root ...'), or they create a menu to be used further (i.e. 'Cut & Paste'). The other buttons are those in the individual windows. These buttons operate on the composite object displayed in the window.

1.4 Future plans for LINCKS

In subsequent releases of the **LINCKS** system, we hope to provide some of the following features which are either missing or incomplete.

- Expanded use of command history. The current implementation of **LINCKS** includes command history (which can easily be disabled at compilation time), but this history is not used to any great extent. We expect eventually to employ command history for complex reasoning tasks.
- Propagate the difference between two object versions to a different (third) object. This application was developed as part of a study in propagation of change.
- A programmer's manual. We hope that other researchers might find the system a useful base upon which to do research. Our emphasis has always been to back theoretical work with implementation experience. The programmer's manual will provide the tools necessary for plugging your application into the **LINCKS** system. If you wish to have preliminary versions of this prior to the next release, please contact us.
- An external application interface, whereby external processes would be able to interact with the system.
- A LINCKS to HTML Gateway (or a World Wide Web server) to a LINCKS database, in order to enable WWW browsers such as mosaic and lynx to browse LINCKS databases.
- A sample application, such as a reference management system.

Chapter 2

How do I ... (basic questions)

Through this chapter, we will try to outline how to accomplish the most common tasks that you will want to do in *xlincks*. Keep in mind that much of command functionality can be customized using translations (see section 3.11 below), but for our purposes we will assume that you are using the defaults.

NOTE: these are not in any particular order yet.

2.1 ... log in?

Basically all you need to know to log in is where the database resides and what your username and password are on that database. Given this information, you simply issue the command:

xlincks database-dir

and type in your username and password at the prompt.

There are also further command-line options. These are:

- **-D**: Include all possible buttons. This includes some buttons which are not of general interest but rather are used for developers' debugging efforts.
- **-N**: Put no buttons into the windows. Those who are very familiar with the keystroke commands might wish to do this to avoid cluttering up their windows.
- **-S**: Suppress certain kinds of informational messages. Not advisable until you understand what the messages are and when they pop up.
- **-C**: Chain the windows together so you don't have to place each new window. New windows will (usually) be placed below and to the right of the window from which they were expanded.
- **-c**: Use color for parallel editing notifications shading and for the result of the "Compare" of two different versions of a document.

- **-u username -p password**: Use provided username and password to login into a database.
- Some standard X Windows application command line options, which are passed on to X and dealt with in the appropriate manner. X options which require a second argument (such as *-bw*) will not work, but *-rv* will.

Finally, if you use only one database on a regular basis, you can set the environment variable *LINCKSDBDIR* where you normally set environment variables (`~/.cshrc`) to the appropriate directory and start *xlincks* without any arguments at all.

setenv LINCKSDBDIR /some/path/to/the/db/directory

2.2 ... log out?

Click on the Command Menu button 'Logout' or type *meta*-l *meta*-q. If there are unstored changes, you'll be asked if you wish to save them.

2.3 ... get to the on-line help?

By convention, on-line help for all of the buttons is available by clicking on the button with the right mouse button. The help texts are stored in the database just as any other data.

Additionally, there is much helpful information in the 'Help!' window which you should get when you log in and which is available by clicking on the Command Menu 'Help Window ...' button. Amongst these are topics such as *M-L keyboard commands*, *Text Editing*, *The Command Menu buttons*, *Glossary of xlincks and LINCKS words*, and more.

2.4 ... expand an item?

Expanding an item can be accomplished in several ways. Keep in mind that 'expanding' means that you are creating a view by combining the current item (as the root of the view) with a GPD used when expanding an item of that type (see chapter 8 for a full description of expansion). Given that you understand that, you have two choices for the kind of expansion. You can either expand the item in a new window or reuse the same window. To expand in a new window, you can either type *meta-l meta-e* with the cursor in the item you want to expand or you can click on the item using the left mouse button while pushing the *ctrl-* button.

To expand and reuse the same window, either type *meta-l meta-E* (note that it's a capital E) with the cursor in the item you want to expand or you can click on the item using the middle mouse button while pushing the *ctrl-* button.

Both of these assume that you haven't changed the translations for these particular sequences.

Under certain circumstances (when the GPD is so defined, e.g. the 'folder' GPD), when you expand a newly-created item for the first time, the system will pop up a window to ask you which view to use in expanding. This means that you should fill in the 'empty: item' with the name of the GPD that you wish to use in the expansion. This will then be saved and used from that point on unless you change the name of the GPD (see section 3.1)[1] See section 5.5 for further information.

2.5 ... create a new item?

There is a single command for creating new items, whether the new item is conceptually 'flat' (those without subitems) or 'deep' (those with subitems). For example, a new paragraph might be considered flat and a new section deep.

At the simplest level, you select the preceeding item which is of the same type as the new item you wish to create. You then type either *meta-l meta-i* (to place it after current position) or *meta-l meta-I* (to place it before current position).[2]

This will create the empty template for the item and put it in the appropriate place. For example, if you wish to create a fourth paragraph, you simply click on the third and type *meta-l meta-i* (or click on the fifth and type *meta-l meta-I*).

For 'flat' items, this is sufficient. For 'deep' items it is slightly more complicated. If you wish to create a new subsection, then you have to indicate that you want a new 'deep' item. You do this by selecting an item which is conceptually at the level of the subsection (the title, for example) and issuing the *meta-l meta-i* command.

Note that both variants on the command will create the item and put in a placeholder if necessary. See sections 5.3 and 5.4 for a description of placeholders.

2.6 ... delete an item?

Simple. Given that you have either clicked on the right item or the cursor is there, type:

meta-l meta-r

This will remove the item and any subitems to that item. For example, if your currently selected item is the title of a subsection, the whole subsection will be deleted. If the item that you wish to delete cannot actually be deleted (as in a singular attribute value), it

[1] The folder GPD is written this way so that each item in the folder can easily use a different GPD in expansion.

[2] Note that there are occasions when the newly inserted item will always be the the first item in its sequence. This is dependent upon the type of item that you insert. If it is a link to another object, it can be ordered. Link tags and attribute tags cannot be ordered, and obviously neither can attribute values (there's only one of a particular attribute value). This occurs because of a very early design decision not to order tags and cannot be corrected without major changes to the model.

will be nullified instead.

2.7 ... edit an item?

This requires a basic knowledge of *emacs*-like editors. The number of commands that you really need is minimal, but we'll give the whole list here:

ctrl- Editing Commands

Command	Description
ctrl-a	Go to beginning of the line.
ctrl-b	Go backward one character.
ctrl-d	Delete character under cursor.
ctrl-e	Go to end of line.
ctrl-f	Go forward one character.
ctrl-h	Delete previous character.
ctrl-j	Split the paragraph into two (when possible).
ctrl-k	Delete to the end of the line.
ctrl-l	Refresh the window.
ctrl-m	Insert a newline.
ctrl-n	Go to the next line.
ctrl-o	Insert a newline but remain on same line.
ctrl-p	Go to the previous line.
ctrl-r	Search backward for subsequent keystrokes.
ctrl-s	Search forward.
ctrl-t	Transpose this character with predecessor.
ctrl-v	Go forward one 'page'.
ctrl-w	Kill selection highlighted by the mouse.
ctrl-y	Insert kill buffer at cursor position.
ctrl-x *ctrl*-s	Save window.
ctrl-x *ctrl*-c	Close window.
ctrl-z *ctrl*-i	Iconify window.
ctrl-z *ctrl*-l	Lower window.
ctrl-z *ctrl*-r	Raise window.

meta- Editing Commands

meta-b	Go backward one word.
meta-d	Delete next word.
meta-D	Delete next word. Put in kill buffer.
meta-f	Go forward one word.
meta-h	Delete previous word.
meta-H	Delete previous word. Put in kill buffer.
meta-j	Join this paragraph with the next.
meta-k	Kill to the end of the paragraph.
meta-q	Reform the paragraph.
meta-v	Go back one 'page'.
meta-y	Insert mouse buffer. As middle mouse button.
meta-<	Go to the beginning of the window.
meta->	Go to the end of the window.
meta-[	Go backward a paragraph.
meta-]	Go forward a paragraph.
meta-Delete	Kill the previous word into kill buffer.
meta-BackSpace	Kill the previous word into kill buffer.

That should get you going.

2.8 ... store my changes?

There are several ways to do this. In a window, you can either click on the 'Store' window button or you can type *ctrl-x ctrl-s*. If you wish to store all of your changes in all views, click on the Command Menu button 'Store All' or type *meta*-l *meta*-s.

2.9 ... undo my changes?

This is perhaps more complicated than it should be. You can only undo back to the last store, and not to a previous transient version (see the definition of transient in chapter 8). For this and other reasons, it pays to store regularly. To undo back to a previous version, simply browse the history (see section 3.5) and find the version that you want to go back to. Then click on it and on the 'Promote' button and store your changes. This view will then be the latest version. You can do similar things on the item level (see section history-item).

2.10 ... move some text from one item to another?

The answer to this depends on how much you want to move. If you want to move text which is all within one item, then keep reading. If, on the other hand, you want to move information which crosses item boundaries, this gets into structure editing since some database structure has to be managed. See section 3.10 for further information about structure editing.

There are two ways to move text from one item to another. The first option is to use

X-Windows cutting and pasting. To do this, simply click with the left mouse button at the beginning of the text you want to move, hold the button down, and drag the mouse to the end of the text. This text is now in a buffer and can be pasted in by moving the mouse into the item where you want to paste the text[3] and clicking once with the left mouse button to get the cursor in the right place, then clicking with the middle button to paste the text. This is the same as other X-windows applications handle this.[4]

The second method is to use the Emacs-like deletion commands that put the killed text into a kill buffer. These commands are *ctrl-k*, *meta-k*, *meta-D*, and *meta-H* (see section 4.3 if you don't know what they do). As long as these are done in sequence, the killed text will be appended to the buffer. As soon as some other command is issued, appending will stop, but the buffer will remain until another kill operation reinitializes it. You can then use *ctrl-y* to 'yank' the text into the item where the cursor is at that moment.

It is important to understand that this doesn't work across item boundaries.

2.11 ... yank back something I've deleted?

If you deleted it using a command that puts it into the kill buffer, you can yank it back using *ctrl-y*. See section 2.10 for further elaboration.

2.12 ... insert text from outside the database?

The answer to this depends upon how much you wish to insert. The simplest way is to use X-Windows cutting and pasting facilities. You simply select the text in some other X window (an *xterm* or whatever) that you wish to insert in an item and insert it there. This is done in the standard X-Windows cut and paste fashion described in section 2.10.

2.13 ... use X's normal cut and paste?

See section 2.10.

2.14 ... know what can be expanded?

There's not really a good answer for this. Whether you can expand something depends on the GPD being used for the view. The best way to gain understand of this is to experiment.

[3] This can be in the same or a different window.

[4] Dragging is not the only way to select text. Double clicking with the left button on a word will select that word, and triple clicking on a line will select the whole line.

2.15 ... duplicate a window?

Simply click on the window button ‘Dup’. This will create a window with the same view and contents. This is simply another way of re-expanding the item that was expanded to create this window.

2.16 ... know if another user is editing an item?

The **LINCKS** system will notify you. When some part of an information component is edited, the database marks that information component as edited by that user. If another user then retrieves that object, s/he is notified that the object is already being edited (see figure 2.1). If the second user then begins editing the same object, the first user will then receive notification that another user has started editing some object s/he was already editing (see figure 2.2). In both cases, the background of the items involved is shaded to indicate which object it is (see figure 2.3).

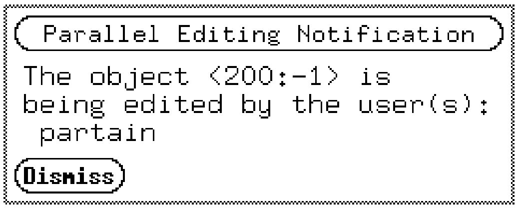

Figure 2.1: ‘Another user is already editing’ notification

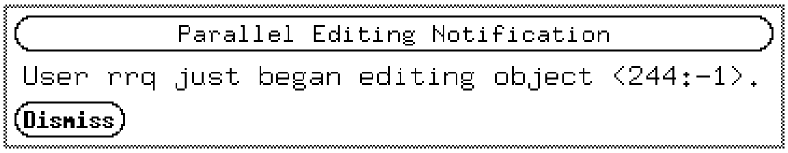

Figure 2.2: ‘Somebody started editing one of your edited objects’ notification

2.17 ... know which GPD’s are available?

The easiest way is to click on the Command Menu button ‘Expand ...’, which will pop up a menu with all accessible GPD’s.

2.18 ... know which GPD is being used for a view?

Simply click on an item in the window and look at the trace output in the window in which you started *xlincks*. The output will look something like:

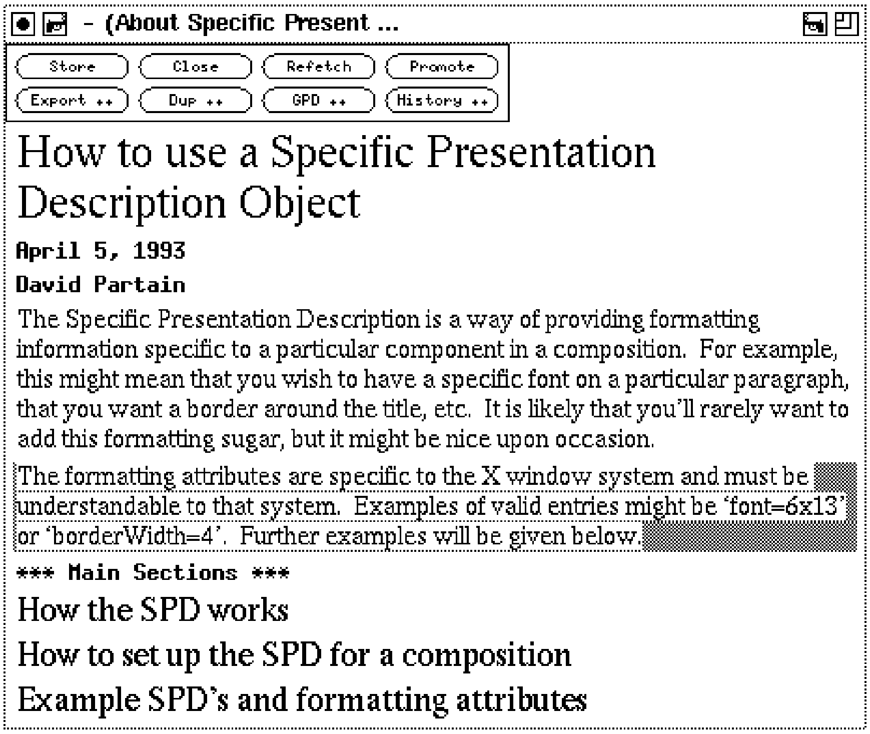

Figure 2.3: Window shading for parallel editing

```
papercopy.main title
      (   0) <380:-1> ATTRVALUE [DOCUMENT:Title] = ...
```

which means the `papercopy` GPD is being used in this view.

2.19 ... reuse the same window when expanding?

See section 2.4

2.20 ... go back to the previous window?

On occasion, you may wish to 'pop' back to the window from which this view was expanded. Conceptually, you can visualize expansion as adding this window to the top of a stack of windows. If you wish to pop off the top window from that 'stack', you can do so in two ways — pop back using the same window and pop back but put it into a new window.

To pop back into a new window, simply type *meta-l meta-p* anywhere in the window.

To pop back into the same window, type *meta-l meta-P* or click with the right mouse button anywhere in the window.

2.21 ... dump an object to an external file?

If you want the textual contents to be written to an external file, simply click on the window button 'Export' or type *meta-l meta-l*. This will create a window with various options about how to export the window. The default is that the window is exported as LaTeX to a file in your home directory. Some LaTeX document parameters can be set as well. Optionally, you can export the window as straight text. In the future, we hope to add further options.

2.22 ... get back my user root object?

Your user root object is your conceptual 'home directory' in the **LINCKS** database. If you accidentally close that window, or if you close it and later decide you want it back, simply click on the Command Menu button 'User Root ...'.

2.23 ... share an object in different views?

This is done by linking in the shared component into the composite objects that you wish to have shared access. The mechanics of this are: click first on the component you wish to link in (source), click second on the component after[5] which you wish to link it (destination), and click on the Command Menu 'Add Link' button.

For example, you could use this mechanism if you wish to have a single 'author' object which will be linked in to all documents you produce (rather than duplicating the information in several 'author' objects). Similarly, if two users wish to work cooperatively on a document, memo, etc., they can share it via a link.

Note the results of this action: a link to the source component is put after the destination component in destination composite object. It does **not** put a link within the destination object that you would then see if you expand that object. That is to say, if you click on a source paragraph in window A, a destination paragraph in window B, and then on the Command Menu 'Add Link' button, the source paragraph will be linked in after the destination paragraph in window B. No link will be put into the object holding the destination composition component.

2.24 ... split one item into two?

After moving to the position where you want to split the item, type *ctrl-j*. The item will be split into two items. If splitting fails, the machine will beep at you. This is because there is no place for the new item to be pasted in. For example, if you try to split a singular attribute (such as a document title in 'full document' view), there is no place for an item between the title and the date so splitting will fail.

[5]There is no mechanism currently in place for adding a link before a component.

2.25 ... join two items together into one?

Type *meta-j*. This works in two ways. If the second item can be removed, its text will be concatenated onto the first item and the second item removed. If it cannot be removed (e.g. a singular attribute value), the text will be concatenated onto the first item and the text of the second item will be nullified.

2.26 ... expand an item in a view other than the default view?

All views are built using a GPD (Generic Presentation Descriptor: see chapter 8). For example, you might have a view of a document created using the GPD 'document overview'. (To know which GPD a view uses, see section 2.18.) Perhaps you wish instead to see the same composition but using the 'full document' GPD. First, click on the Command Menu button 'Expand ...', which will create a menu of all available GPD's. Then click on the item you wish to expand and click on the name of the GPD you wish to use. This will expand the item using that GPD.

One useful GPD to use in this way is the 'node' GPD, which gives a representation of the entire information component of which the item is a part.

2.27 ... close all windows?

You can close all windows by either clicking on the Command Menu button 'Close All' or by typing *meta*-l *meta*-c. If there are any changes, you'll be asked if you want to save them.

2.28 ... close all unchanged windows?

Do the same as is described in section 2.27, but click on 'Cancel' when asked if you want to save changes in windows that have changed.

2.29 ... iconify, raise, or lower a window from the keyboard?

To iconify the window, type *ctrl*-**z** *ctrl*-**i**, to raise the window *ctrl*-**z** *ctrl*-**r**, and to lower the window *ctrl*-**z** *ctrl*-**l**. This is really window manager functionality, but it comes in handy for people who don't like to take their hands off the keyboard to operate the mouse.

2.30 ... add a link to another object?

See section 2.23.

2.31 ... close all these auxiliary windows?

You can close all at once by either using the Command Menu 'CloseAuxWins' button or by typing *meta*-l *meta*-w.

2.32 ... change my password

The *dbpasswd* program is used for this. Type:

```
dbpasswd username
```

where `username` is your **LINCKS** username. This assumes that you have the **LINCKSDBDIR** environment variable set to the directory holding the **LINCKS** files. If you don't, use the `-d database-directory` option. This is explained fully in the *dbpasswd* man page.

Chapter 3

How do I . . . (advanced questions)

3.1 . . . change the default GPD used in expansion?

Expand the item using the 'define expand types' GPD. Change the name of the GPD used for expansion to the that GPD you wish to use instead. For example, from figure 3.1 to 3.2 the default GPD was changed from 'folder' to 'Myfolder'.

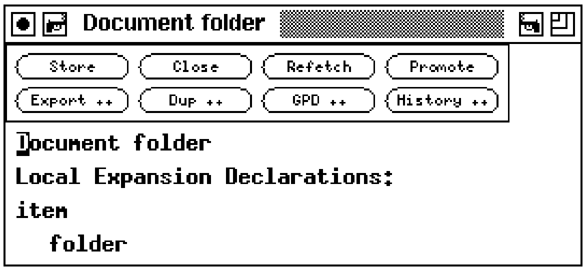

Figure 3.1: Before changing default GPD

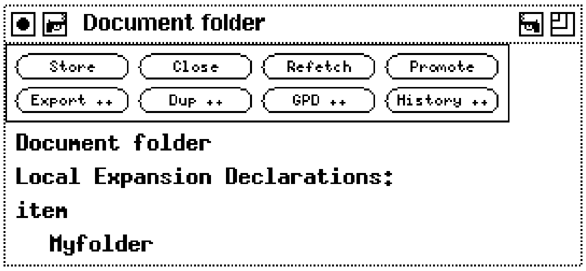

Figure 3.2: After changing default GPD

3.2 . . . change which windows I get at login?

By default, you get the 'Help!' window, the 'Simon Says...' window, and your user object when you log in. We recommend that you alter this only after becoming very familiar with the system.

When you want to change which windows you get (the 'login profile'), expand your user object using the 'login profile' GPD (see section 2.26). That should bring up a window similar to figure 3.3.

Click on one of the GPD names (e.g. 'text' or 'user') and create a new item (*meta-l meta-i*). This will give you figure 3.4.

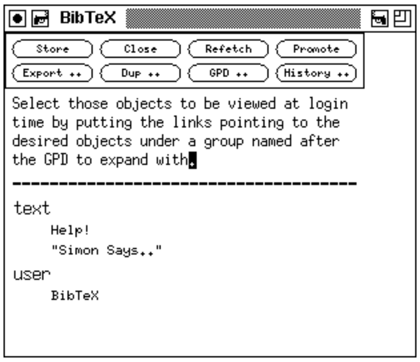

Figure 3.3: Login profile prior to changing

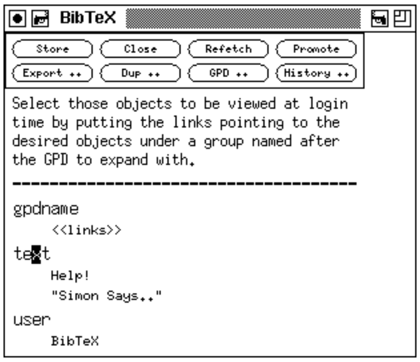

Figure 3.4: Login profile after adding new field

Change 'gpdname' to the name of the GPD to be used in expansion and link in the object that you wish to expand where <<links>> is. You will then get a window with that object when you log in. The completed login profile will look like 3.5.

If, instead, you wish to add another window but using a GPD already in the list ('text' or 'user' in this case), you can simply link in the new object after one in the list. For example, if you wish to open the document in 'text' view rather than 'document overview', it would have looked like figure 3.6.

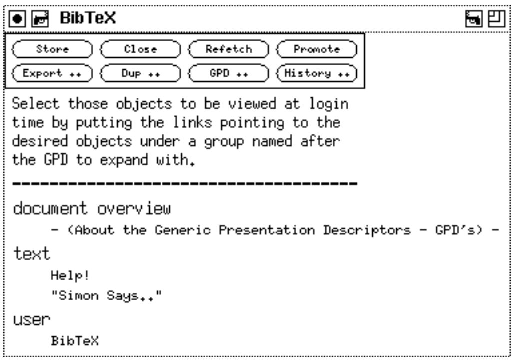

Figure 3.5: Completed login profile changes

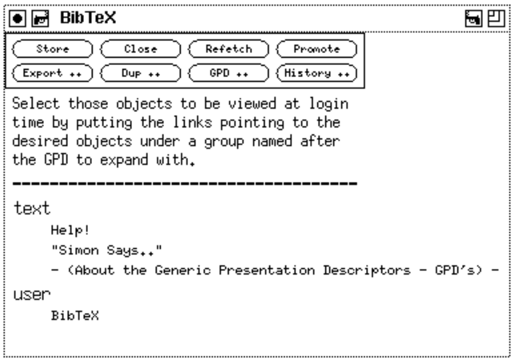

Figure 3.6: Adding new object using existing GPD

3.3 ... refetch something already in my workspace?

The short answer is to click on the 'Refetch' window button. This will re-retrieve the latest version of the composite object shown in this window.

Clicking on the 'refetch' button causes a merge of the versions of the object with any new versions that have been stored since the object was last retrieved. If there has been a change, there are three cases to be considered:

- You have not made any changes, but someone else has made changes in the composition and stored them. In this case, the latest composition will become visible.
- You have unstored changes to a component but other users' stored changes are to different components in the composition. All the newly stored changes will become visible.
- You have unstored changes to a component and there are newly stored changes to the same component. Your transient version of that component will remain visible,

but the stored version will be merged into the history.

In all cases, the merged versions are accessible through the history browsing. (See section 3.5 and 3.6 for information on history.)

A bit of explanation of why you might want to do this:

As you are editing an object in your workspace, all changes stay only in the workspace until you store the object. Until you store, this version of the composite object (window) is called transient (see the glossary in chapter 8).

Additionally, while you are editing, another user might begin editing the same item or another item in the same composite object, thus creating possible conflict.

Once the composite object has been retrieved from the database into the workspace, it will not be retrieved again unless explicitly requested. However, under some circumstances you may wish to explicitly retrieve what the database has as the latest version of the object. For example, you may wish to merge the changes made by another user into the current version of the composite object so as to see the changes.

3.4 ... go back to an old version of a composite object?

See section 3.7.

3.5 ... look at the history of a composite object

If you want to browse through the history of a composite object (look at old versions), either click on the window button 'History ...' or type the command *meta-l meta-h*. This will create a window with circled numbers, which are the previous versions of the composite object. Clicking in a circle makes the view using the old versions of the the components and makes the window uneditable. If you wish to return to the latest version, simply click on the leftmost circle (this also make it editable again). Figure 3.7 gives an example of such a history graph. The leftmost circle represents the logical composition, whereas the other circles represent a version of the composition. If, as in figure 3.8, the word 'hidden' appears, that means that there are previous versions, but not all are shown in the graph. Clicking on 'hidden' will fetch the set of versions immediately prior to the version to the right of the word 'hidden' (in figure 3.8, the set of versions prior to version 895).

3.6 ... look at the history of a single item

In addition to composite object history, you can look at how a single item has developed. To do so, make sure that focus is in the correct item by clicking there and click on the Command Menu 'Item History ...' button. That will pop up a window with circled numbers. Clicking in the circles will put that version of the item into the view. Note that

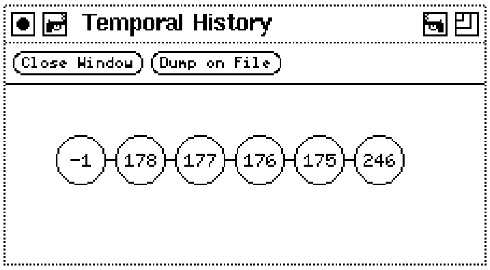

Figure 3.7: Composition history graph

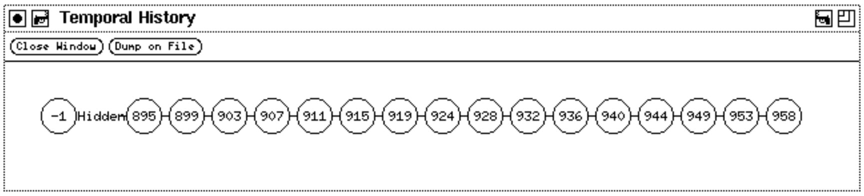

Figure 3.8: Composition history graph with hidden versions

when you do this, the view ceases to be editable, but that it does change the composite object. If you want to promote an old version to be the latest version (see section 3.8), use the 'Promote' button.[1] In order to make the view editable again, bring up the composition history (section 3.5) and click on the leftmost circle. Figure 3.9 shows an example of an item history with parallel development (two users worked on the component at the same time).

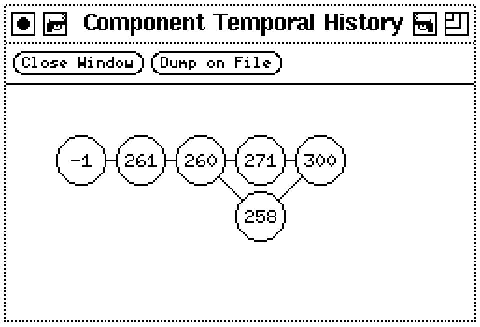

Figure 3.9: Item history graph

3.7 ... promote a composite object?

Recall from chapter 8 that 'promote' means to upgrade an old version to be the latest version. In the case of a composite object, this is done by first browsing the history

[1]The window title on this window is 'Component Temporal History'. Don't let this confuse you. A 'component' refers to an entire information component from which a part might be taken for the item. From the user's perspective, however, browsing through the history will be as if they are browsing through the history of that item.

(section 3.5) and deciding which old version you wish to promote. Then you simply click on the window button 'Promote', which will make that old version visible in the window into the latest version.

3.8 ... promote an old version of an item?

This works in much the same way as promoting a composite object. First browse the item history and choose the one you want by clicking in the appropriate circle (see section 3.6). Note that each click will promote that item to latest, so simply choose the one you want last. You can make the view editable again by clicking on the first circle.

3.9 ... compare two different versions of a composition?

This is done using the Command Menu 'Compare' button, which does a structural difference between two versions of the the same composition. You need to bring up two different versions of the composition on the screen. Currently, the easiest way is to make another copy the current version (see section 2.15), which we call the 'destination' version. Then go back to an old version of the composition (see section 3.5) in the duplicate window.

Select the source (the old version), then the destination (the current version), and finally click on the 'Compare' button in the Command Menu.

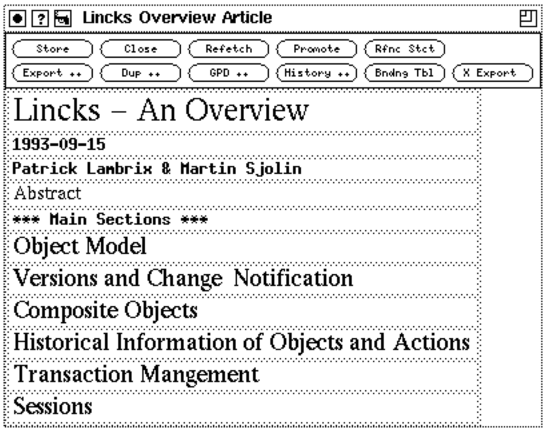

Figure 3.10: Source Composition (Original version).

That is:

- First, click on the 'Dup' window button to get a duplicate of the window.
- Secondly, in one of those, bring up the history of that composition by clicking on the 'History' window button. Browse the history in the manner described in section 3.5.

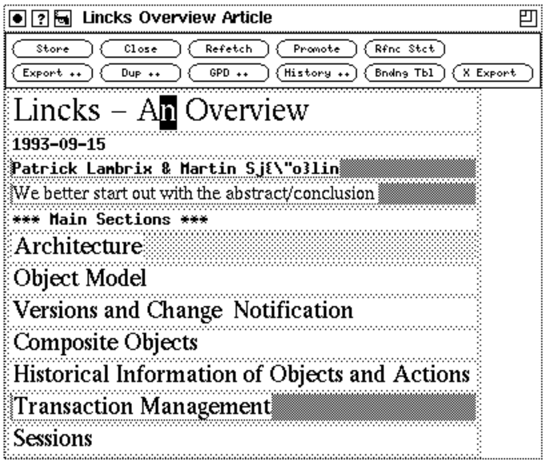

Figure 3.11: Destination Composition (New version).

- After determining which older version you want to use in the comparison, click first (anywhere in the window) in the old version, then in the newer version.
- Click on the 'Compare' button in the Command Menu.

Deleted entities in the **source** composition (where you clicked first) will be marked with shaded background or with color background (if you gave the '-c' command line option). Where there have been changes from the source to the destination, these items will be marked in the destination composition by shading. Items which have been added will also be so marked. You can customize the select backgrounds; see section 3.15.

Figure 3.10 is selected first and thus is the 'original version' and figure 3.11 is the 'new version'. In figure 3.11 we have inserted a new section, "Architecture", which has been shaded light grey, and the top level changes to the name field and the abstract part is shaded in a darker grey.

To refresh the view and remove the coloring, simply use the *ctrl-l* key.

3.10 ... move a block from one view to another?

This is done using the Command Menu 'Cut & Paste' button, which implements what we call 'structure editing'. This is editing which conceptually crosses an item boundary, such as deleting multiple items, cutting several items from one location and inserting them elsewhere, and so forth.

The mechanics are simple. First, bring up the structure editing menu by clicking on the 'Cut & paste ...' button, which will bring up a window menu which looks like figure 3.12.

Select the beginning and end of the area you want to copy or cut out (by clicking at the beginning and ending item using the left mouse button) and select either 'Cut Region' or 'Copy Region'. Thereafter, select the position where you want to paste the block and use either the 'Paste after' or 'Paste Before' in the Cut & Paste menu.

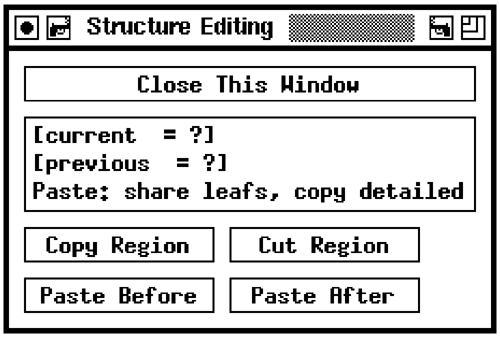

Figure 3.12: Cut & Paste menu window

There is one additional field of interest in the menu: the field beginning with *Paste:*. By clicking on this line, you toggle through a series of options as to how the structure editing will behave. For the most part, structure editing tries to share objects as much as possible. However, if you truly want a copy that you can edit without affecting the source, you can change this line (by clicking on it to toggle) so that you can do so. The default option is to share the leaf items but copy inner items object-wise (option 1 below).

These options are:

1. **share leafs, copy detailed**: means that the leaf objects (the visible ones) are shared in both the source composition and the destination where you paste it. It doesn't, however, copy the parts of the visible objects which are not visible (i.e. attribute values which aren't seen).
2. **share leafs, copy whole objects**: again, this means that leaf objects will be shared between the source and destination. Objects that are not shared but copied are copied as a whole, including eventual invisible attributes and links.
3. **share all, copy detailed**: means to always avoid copying when possible and instead actually use the same object. However, if the destination structure differs from the source structure so that some source object would need to be changed, then that object is copied rather than shared. Copying that must occur is detailed, i.e. only visible items are copied.
4. **share all, copy whole objects**: means the same as option 3, except that also the invisible items are copied when copying occurs.
5. **share none, copy detailed**: means that the source is copied into an entirely new object structure and only the visible items are copied.
6. **share none, copy whole objects**: means that the source is copied into an entirely new object and that all parts of the source object are copied — visible or not.

Caveats:

- Cut & Paste operates only on complete items, so it makes no difference where you click in the item. If you wish to use only part of an item in the move, first split the paragraph (using *ctrl*-j) and proceed from there.

- Structure editing is difficult to automate, and as such may not always behave in the way that you find most appropriate. We have tried to match its behaviour with what we envision the user would want, but such guesses are subjective. We would appreciate descriptions of circumstances under which the structure editing behaves in ways that are unexpected.
- By default, either pasting will share as many objects from the source composition as possible. In other words, if you copy a paragraph and paste it in another composition, it will be a link to exactly the same component. However, this behaviour can be changed by clicking on the line which begins 'Paste:' as outlined above. Clicking so that you choose the option **share none, ...** will make a copy which you can then alter without altering the source.
- It is unwise to use structure editing without understanding the underlying structure of the composite object that you are editing. Although read-order is preserved, since structure editing does involve very specific structure in both the source and destination, there may be side-effects that are predictable but perhaps difficult to explain if you don't understand the underlying database structure. For example, if you move a block between very differently structured objects (such as a document and a GPD), you can expect odd results. Of course, you may want those odd results ...
- Finally, the structure editing is to be considered in beta testing. It might be buggy.

3.11 ... have my own set of X translations?

If you don't know what translations are, you probably don't need to do this. However, it is possible to attach certain actions within *xlincks* to particular keyboard sequences. For example, the default keyboard sequence for expanding an item in a new window is `Ctrl<Btn1Down>` and the default for expanding in the same window is `Ctrl<Btn2Down>`. If, instead, you wish to switch these, you can put the following in your *.Xdefaults* file:

```
! This line, begun with ! is a comment line
! switch the normal translations
xlincks*Contents.?.Translations: #override\
  Ctrl<Btn2Down>: exp-in-new()\n\
  Ctrl<Btn1Down>: exp-in-place()



! bind these actions to L? buttons to make them easier to use
xlincks*Contents.?.Translations: \#override\
  <Key>L1: pop-in-place()\n\
  <Key>L2: pop-in-new()\n\
  <Key>L4: exp-in-new()\n\
  <Key>L3: exp-in-place()



! bind these in a really strange place
xlincks*Contents.?.Translations: \#override\
```

```
Meta<Key>1: store-window()\n\
Meta<Key>2: close-window()\n\
Meta<Key>3: insert-plural()\n\
Meta<Key>4: delete-plural()
```

3.12 ... know what the possible translations are?

The following translations can be installed in the manner described in section 3.11.

`close-widls()`	Close the auxiliary windows.
`close-all-windows()`	Close all windows.
`close-window()`	Close the window.
`compare-versions()`	Difference between the previous and current selected composition.
`delete-plural()`	Remove the current item and its subitems.
`exp-in-new()`	Expand the current item in a new window.
`exp-in-place()`	Expand the current item in the same window.
`export-window()`	Export the window as LaTeX or text to an external file.
`iconify-win()`	Iconifies the window.
`insert-plural(before)`	Create a new item of the same type as the current item and place it before current item.
`insert-plural(after)`	Create a new item of the same type as the current item and place it after current item.
`join-text()`	Add the text from the subsequent item to the end of the item which currently has the cursor.
`logout()`	Do just that.
`lower-win()`	Lowers the window to be on below any windows which overlap it.
`pop-in-new()`	Revert to the previous window into a new window.
`pop-in-place()`	Revert to the previous window in the same window.
`promote-version()`	Promote an older version of a composite object to be the latest.
`raise-win()`	Raises the window to be on top of any windows which overlap it.
`refetch-latest()`	Refetch the composite object from the database.
`show-gpd()`	Show the GPD used for this view.
`show-history()`	Show the composite object temporal history for this view.
`show-node-history()`	Show the individual object temporal history.
`split-text()`	Split the paragraph at the cursor position, placing the text which comes after the cursor at the beginning of the next item.
`store-all-windows()`	Store all changes in all windows.
`store-window()`	Store changes to the window.

There are, of course, other translations, but these relate to the editing functions of the system and probably should not be altered.

3.13 ... dump an object so it can be loaded into another database?

See chapter 6 for complete details.

3.14 ... write my own GPD?

Chapter 7 beginning on page 45 deals with this subject.

3.15 ... change application specific resources?

You can customize certain behaviour through the use of application defaults, which can be placed in your *.Xdefaults* file. You have two main categories - setting the defaults values used when exporting in LaTeX format (see section 2.21); specifying backgrounds for Parallel Editing Notifications and difference operations.

For example, to set the LaTeX export to default to **article** style and use colors when displaying parallel editing notifications, add the following to your *.Xdefaults* file:

```
xlincks*exportStyle: article
xlincks*useColors: True
```

Application resources for LaTeX export:

Resource Name	Type	Default	Values
`exportLatex`	Boolean	True	True, False
`latexStyle`	String	article	article, letter
`latexDeep`	Boolean	True	True, False
`latexFontSize`	Integer	11	10, 11, 12
`latexPlaceHolders`	Boolean	True	True, False
`latexEscape`	Boolean	True	True, False

The following resource are used if you elect to use colors (either by command line option '-c' or using the `useColors` application resource). They are all of resource type *Pixel*:

Resource Name	Default
`pixelPen`	red
`pixelChange`	yellow
`pixelDelete`	green
`pixelInsert`	green
`pixelInvisible`	orange

By default, *Pixmaps* are used for shading different backgrounds. When specifying another Bitmap, you *must* specify the width and height which are given in pixels. The bitmaps value is a path to a bitmap or a name of a bitmap which can be found in the standard places. The following are the names of resources and their default values:

Resource Name	Default
`pixmapPen`	gray
`pixmapPenWidth`	2
`pixmapPenHeight`	2
`pixmapBound`	gray3
`pixmapBoundWidth`	4
`pixmapBoundHeight`	4
`pixmapChange`	gray
`pixmapChangeWidth`	2
`pixmapChangeHeight`	2
`pixmapDelete`	gray
`pixmapDeleteWidth`	2
`pixmapDeleteHeight`	2
`pixmapInsert`	gray3
`pixmapInsertWidth`	4
`pixmapInsertHeight`	4
`pixmapInvisible`	gray1
`pixmapInvisibleWidth`	2
`pixmapInvisibleHeight`	2

Chapter 4

Summary of xlincks commands

4.1 The 'Command Menu'

Button name	Description
Expand ...	expand an item with an alternate GPD.
Cut & Paste ...	bring up a menu to be used for structure editing.
Item History ...	show a graph of the history of a particular item.
User Root ...	rebuild the user root ('home directory').
Help Window ...	build the 'Help!' window for access to on-line help.
Add Link	add a link to an object.
DBDump	dump a t2lincks file of a part of the database.
Store All	store all changes.
Close All	close all windows.
Logout	yes, logout.
CloseAuxWins	close auxiliary windows.
Compare	compare two versions of a composition.

4.2 Buttons in each window

Button name	Description
Store	store changes in this window.
Close	close this window.
Refetch	fetch any changes since last retrieved.
Promote	promote this version to be latest.
Export ..	export window as a text file.
Dup ..	duplicate the window.
GPD ..	show the GPD used for building this view.
History ..	show the history graph of this composition.
Rfnc Stct	(reference structure) show a graphical representation of the internal representation of the view.
Bndng Tbl	(binding table) show the binding table, the mechanism used to manage composite objects.

4.3 Emacs-like commands

Command	Description
ctrl-a	move to the beginning of the line.
ctrl-b	go back one character.
ctrl-d	delete next character.
ctrl-e	move to the end of the line.
ctrl-f	go forward one character.
ctrl-g	reset.
ctrl-h	delete previous character.
ctrl-j	split the paragraph.
ctrl-k	kill to the end of the line.
ctrl-l	redraw and refresh the display.
ctrl-m	add a newline.
ctrl-n	move to the next line.
ctrl-o	open a new line but stay on the current line.
ctrl-p	move to the previous line.
ctrl-r	search backward.
ctrl-s	search forward.
ctrl-t	transpose characters.
ctrl-v	move to the 'next page'.
ctrl-w	Kill selection highlighted by the mouse.
ctrl-y	Insert kill buffer at cursor position.
ctrl-x ctrl-s	save the window.
ctrl-x ctrl-c	close the window.
ctrl-z ctrl-i	iconify the window.
ctrl-z ctrl-l	lower the window.
ctrl-z ctrl-r	raise the window.
meta-b	move backward one word.
meta-f	move forward one word.
meta-k	kill to the end of the paragraph.
meta-q	format the paragraph.
meta-v	move to the previous page.
meta-d	Delete next word.
meta-D	Delete next word. Put in kill buffer.
meta-h	Delete previous word.
meta-H	Delete previous word. Put in kill buffer.
meta-j	join the text of this and succeeding item.
meta-y	Insert mouse buffer. As middle mouse button.
meta-<	move to the beginning of the window.
meta->	move to the end of the window.
meta-]	move forward a paragraph.
meta-[	move backward a paragraph.
meta-Delete	Kill the previous word into kill buffer.
meta-BackSpace	Kill the previous word into kill buffer.

Command	Description
meta-l meta-b	promote this version of the view.
meta-l meta-c	close all windows.
meta-l meta-d	difference between two versions of a composition.
meta-l meta-e	expand this item in a new window.
meta-l meta-E	expand this item in the same window.
meta-l meta-p	pop back to previous window in a new window.
meta-l meta-P	pop back to previous window in the same window.
meta-l meta-g	show the GPD used to build this view.
meta-l meta-h	show the composite temporal history.
meta-l meta-i	insert a new plural item after.
meta-l meta-I	insert a new plural item before.
meta-l meta-l	export the window.
meta-l meta-o	show the item temporal history for current item.
meta-l meta-q	logout.
meta-l meta-r	delete this plural item.
meta-l meta-s	store all changes in all windows.
meta-l meta-u	refetch the latest version of this view.
meta-l meta-w	close auxiliary windows.

4.4 Command Equivalencies

This table [1] shows equivalent ways to use the functionality available in the window buttons, either by using the button, or by using an *emacs*-ish command.

Command Description	window button	*emacs*-like
Store a window	Store	*ctrl-x ctrl-s*
Close a window	Close	*ctrl-x ctrl-c*
Refetch a composition	Refetch	*meta-l meta-u*
Promote an old composition	Promote	*meta-l meta-p*
Textually export a window	Export ..	*meta-l meta-l*
Duplicate a window	Dup ..	*meta-l meta-l*
Show the Generic Presentation Descriptor used for this view shown in this window	GPD ..	*meta-l meta-g*
See the history of the composite object shown in the window	History ..	*meta-l meta-h*
Show the internal structure used to create this view shown in this window	Rfnc Stct	—
Show the binding table for the composite object shown in this window	Bndng Tbl	—

This table shows equivalent ways to use the functionality available in the Command Menu buttons, either by using the button, or by using an *emacs*-ish command.

Command Description	button name	*emacs*-like
Expand an item using a different GPD than is normally used.	Expand ...	—
Perform structure editing.	Cut & Paste	—
Look at the history of a single item.	Item History ...	*meta-l meta-o*
Rebuild the user root object.	User Root ...	—
Add a link to some object.	Add Link	—
Dump some portion of the database to a t2lincks file.	DBDump	—
Store all changes.	Store All	*meta-l meta-s*
Close all windows.	Close All	*meta-l meta-c*
Logout of *xlincks*.	Logout	*meta-l meta-q*
Close auxiliary windows.	CloseAuxWins	*meta-l meta-w*
Compare two versions of a composition	Compare	*meta-l meta-d*
Propagate	Propagate	—

[1] *ctrl-* will be used in the table to symbolize the 'control' key. *meta-* will be used for the 'meta' key.

Chapter 5

Frequently Asked Questions

Note that there is a more exhaustive and current list of frequently asked questions in the top directory of the **LINCKS** distribution. The file is called LINCKS.FAQ.

5.1 How do I close the menu I get by pushing the 'Expand ...' button?

This window is a bit different than all of the other windows that you see when you log in. You get rid of it by clicking on the top of the window where it says "(Select here to close the window)".

5.2 What does it mean when I get a "Couldn't build view using '*somename*"' error message?

After exhausting all the other places that the system looks for how to expand something, it tries to use the name of the item that you are trying to expand. For example, if you try to expand on a title, but the system can't figure out how to do this, the last GPD name it will try will be 'title'. Since there is no such GPD, you'll get an error saying "Couldn't create view using 'title'. No such GPD.'

If you know how you want to expand this item, you can set it so that that GPD will be used in expansion by following the procedure in section 3.1. If you want to expand it just once, you can follow the procedure found in section 2.26.

5.3 What are these items surrounded by << . >>?

This is a kind of placeholder for something which is expected given the Generic Presentation Descriptor used to create the view but which doesn't exist in the database. If you then edit this text, *xlincks* will create the item in the appropriate information component.

5.4 What is this 'empty: ⋯' item?

Again, this is a placeholder. However, this placeholder indicates not that the item doesn't exist, but rather that it is a null string. Again, this can be edited just as you would any other text.

5.5 What should I type when asked what view to use?

This question is relevant if you get a window that looks like figure 5.1.

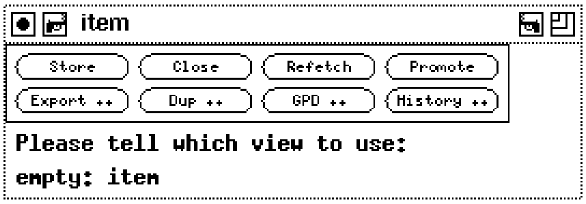

Figure 5.1: Window asking which view to use in expansion

This will only happen when you expand an item for the first time in a view (such as 'folder') which is defined to allow items to be expanded in different ways. Simply fill in 'empty: item' with the name of the GPD you wish to use and re-expand the item. The GPD you just indicated will then be used to create the view.

5.6 What's the difference between cutting and pasting with X and using the structure editing?

The difference is that cutting and pasting with X knows nothing about structure. All of the text must be pasted into a single item, and only text in a single item can be selected. Structure editing attempts to manage the database structure as well. Cutting and pasting with X is described in section 2.10 and structure editing is described in section 3.10.

5.7 What's this information window about transient components?

Normally, when you create a view, if there are components of the composite object which are more recent than the components given in the latest binding table (see chapter 8) for the composite object, the binding table will be automatically updated and stored. However, if the components which are more recent are transient, this is **not** done. The reason for this is that if the binding table were stored with references to transient components and you then did not store those transient components before quitting, the binding tables would reference an object which doesn't exist in the database. The message is simply telling you that the binding table cannot be stored. This message is suppressed by the command-line argument **-S**.

5.8 Why does it ask if I want to save when I haven't changed anything?

You may not have changed anything, but something has changed. This can happen for several reasons, but the most likely one is that creation of some view inserted constants into an information component. In many GPD's, string constants are loaded into the information component the first time the view is created. This happens without any action by the user.

Of course, it is always safest to save changes under these circumstances.

5.9 What is this trace output in the window where I started up *xlincks*?

For the most part, this can be ignored. For most users, the information is uninteresting, but for developers and those wishing to debug, they might provide some assistance.

5.10 What are these notifications about someone else editing something?

This means that some object that you have started editing is being edited by another user (see section 2.16). Note that there are two kinds of objects for which you get messages. One is for binding tables, which basically means some composite object, and the other is for individual items. The former means that there is parallel editing in some composite object but not necessarily on the same component (item). The latter means that the same information component is being edited in parallel.

5.11 What happens if I don't save changes when I log out?

If you don't save changes, there will be no effect upon the database at all. The state of the objects will remain as they were at last store.[1]

[1] This isn't quite true. It is possible that new binding tables will be stored automatically as you browse, but this is as it should be.

5.12 Why is part of my window shaded?

There are two possible reasons. If the shading is very light and covers the entire window, it means that the window is not editable. This is because you have browsed through either item history of one item in the window or the composite object history. If, instead, the shading is quite dark and is (probably) only on part of the window, that means that that object is being edited in parallel with some other user (see section 2.16).

5.13 Why does it dump core when I do ...?

It shouldn't. If you encounter a core dump, please let us know. If you wish to find the bug and send us a patch, that would be great. If you instead wish to send us the output of either `gdb xlincks core` or `dbx xlincks core`, that would be very helpful. In any case, please send mail to `lincks-bugs@ida.liu.se` and tell us what you were doing, if it's reproducible, and the debugger output if possible.

Chapter 6

On Selective Dumping

NOTE: This chapter is exported from the database. It should be changed in the database and exported, not changed here.

Selective dumping is the process of taking a part of a database and writing it out as a t2lincks text file which can be reloaded into this or another database. This is the only method available to extract the database information in a structured manner so that it can be moved to another database.

The selective dumping facility is invoked by the command menu DBDump button, which presupposes a prior selection in a dump file description object. The dump file description object tells which file to dump to as well what the dump file should contain.

6.1 Creating Dump Description Object

In this section, we will go through the steps necessary to create a dump file description object used to dump some portion of the database. In figures 6.1 and 6.2 you'll see an example of such an object and can use it to follow along with the description.

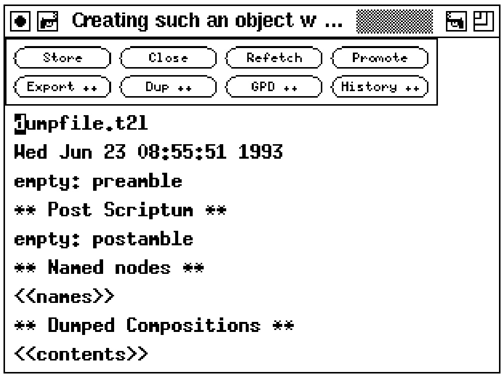

Figure 6.1: A newly created dump file description object

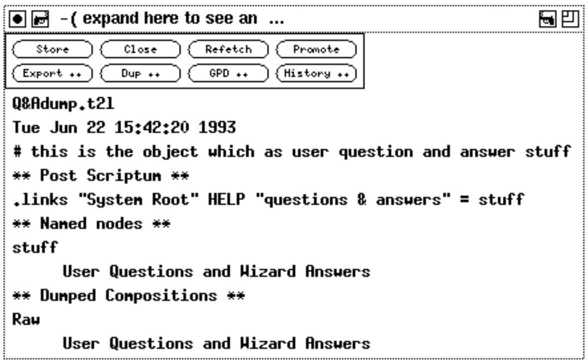

Figure 6.2: A complete dump file description object

1. Create a dump file description object by creating a new item in a folder and expanding it in the 'dump file description' view. The dump file description GPD is available in the Dump GPD map, which is one of the system level gpd maps.

Creating such an object will give you something that looks like figure 6.1.

2. Decide the file name of the dumped t2lincks file that will be written out and put it on the first line. The default name, dumpfile.t2l, may also be left unchanged, in which case it will be used as the filename. You may use a full path name, but can also use names relative to the current working directory.

3. The second line, the date, should be left unaltered.

4. If desired, write an appropriate file preamble by changing 'empty: preamble' to whatever you want. The preamble text is written literally at the beginning of the file, and it is intended to be used for comments (beginning with #) to describe the general contents of the dump file.

5. Next, if desired, write a file 'postamble', text to be written literally to the end of the dump file. Do this by changing 'empty: postamble' to the postamble you want. This text is typically a specification of where in the database the information should be put when reloaded.

A few examples of what this might be are:

The .set command, which aliases a particular node path. For example,

.set myuser from "System Root" +USERS:partain:1

would assign the symbolic name 'myuser' to the node accessed by following the link USERS:partain:1 from the System Root. System Root is a predefined constant. The value of this command is that this symbolic name ('myuser') can then be used in linking in that which you are dumping into a well-known spot in the database.

The .links command, which puts a link to an object in a well-known object. For example,

.links myusers FOLDER Item = dumpobject

would put a link at FOLDER:Item:1 in 'myuser' (defined above) to the object defined as dumpobject in the names section (see number 6 below).

Using the combination of these two commands, you are able to link the dump objects into the database. Otherwise, they will be added but you will have no way to find them!

Note that these commands are t2lincks commands, so you can put any t2lincks command here that you want. See the t2lincks man page for more information.

Note that these commands are t2lincks commands, so you can put any t2lincks command here that you want. See the t2lincks man page for more information.

6. In the <<names>> section, you can predefine symbolic names for specific nodes. Normally only the 'root' nodes of the dumped structure are named explicitly, since you need to know these names in the postamble (to hook in the structure a reload time). Still, you may name any object. All explicitly named objects will generate an '.object...' command.

7. Finally, you list all compositions you want to save in the dump file. This is done in the <<contents>> section by first stating which GPD will be used in the dump and all of the compositions to be dumped under that GPD. This is similar to the way that windows are chosen at log in time. Under a GPD name you put links to all objects to save using that view. Note that this does not recursively descend through the database. If you wish to do this, see the next section.

6.2 Dumping a part of the database recursively

The dump declarations described above are typed dumps, because the objects are dumped under a particular view and the dumped information depends on the chosen GPD's. If, instead of this, you want to dump whole subtrees of nodes in node view, you can do this by listing the subtree root nodes under the GPD tag 'Raw'. Again, this is done in the <<contents>> section of the 'dump description object'. All nodes recursively reachable in node view, beginning at given root nodes, will then be written to the dump file. It is likely that this will be the most common way for you to dump parts of the database.

In conjunction with the 'Raw' tag, you will likely want to use the 'Avoid' tag. By listing nodes under the GPD tag 'Avoid', the subtree dump traversal will not recurse through the indicated nodes. In this way you can inhibit subtree dumping along cross links. Again, this is particularly useful in combination with 'Raw' dumps.

6.3 Extracting the information

Having prepared an appropriate dump file description you select it (click with the left mouse button on the file name, for example), and click on the 'DBDump' button on the xlincks command menu.

The dump file description object is processed in two steps. First each dump composition is inspected and all elementary objects are collected and given symbolic names. The predefined names are of course used if given, otherwise names in the style of node0001434 are generated. During this first step, all indicated compositions are expanded into complete reference structures (which as usual may result in binding table updates). Thereafter the file is written in the following way: 1) a short identification, 2) the preamble text, 3) object creation statements, 4) object contents statements organised according to the specified views, and last 5) the post scriptum.

Note that dumping can be slow, particularly if you dump in Raw mode.

6.4 Miscellaneous comments about dumping

After dumping some portion of the database using this method, you can then load it into another database using t2lincks. See the t2lincks man page for how to do this.

Note also that using DBDump is not necessary if you want to dump the entire database.

Instead, there is a separate program called 'dbdump' which is intended for this purpose. See its man page for further information.

Moreover, when dumping parts of the database where objects have binary image information (not ending with a zero or containing several zero characters), the dumping process might create extra files for the binary image(s). The generated files are named `NODE111.image` and are used in a statement like '`.binary NODE111 < "NODE111.image"`'. The filenames are derived from the object's names with the extension `.image`. Files are *only* generated when the size of the binary information is greater than a certain constant (currently 1000 bytes); in other cases the binary information is included as a uuencoded string in the generated t2lincks file.

Chapter 7

NOTES on Generic Presentation Descriptors

This chapter is brief documentation of the Generic Presentation Description concept implemented in the *xlincks* program. It is taken largely from the *xlincks* system and is available there in the 'Help!' window in much the same form.

7.1 Introduction

The notion of Generic Presentation Description (GPD) is one of the basic notions of the present LINCKS system, and it is essential to know how to write GPD's in order to obtain tailored views. This chapter contains a number of notes intended to give the necessary documentation and basis for a detailed discussion on the notion of GPD's.

A Generic Presentation Descriptor is the template used to build a view of a composite object in a window. Given a root object, the GPD is used to tell which component objects are to be displayed in that view, as well as which parts of those objects to view (attribute, image, etc.). Additionally, the GPD tells how to access the objects, how they are to be expanded if the user does so, and any special formatting requirements for the display.

For example, given the root node 'David' and the GPD 'user' (see figure 7.1), we might build a view representing the user David. We see that the 'user' view is made up of a folder, a separator, and a gpdmap. The 'folder', in turn, is divided into the description, date, another divider, and items. This defines the STRUCTURE of the 'user' view. Then, looking at the ACCESS section, we see that the GPD picks out the 'description' from the IMAGE, the 'date' is taken from an attribute value SYSTEM:Created, and the 'item's are all the linkitems in the link group FOLDER:Item (these are the parts of figure 7.2 with a border around them). Finally, in the last part of the view, the 'gpdmap' is all of the linkitems in the SYSTEM:GPDmaplist link group. The dividers 't1' and 't2' are also attribute values which are given a default value upon creation of the object. A complete view built using this GPD can be seen in 7.2.

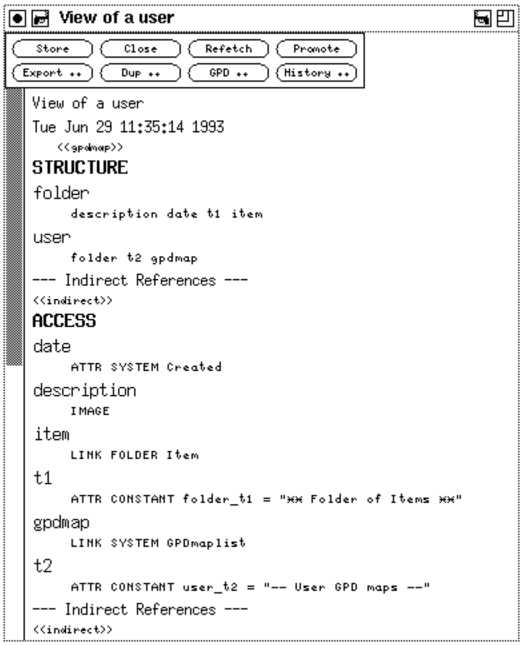

Figure 7.1: The 'user' GPD

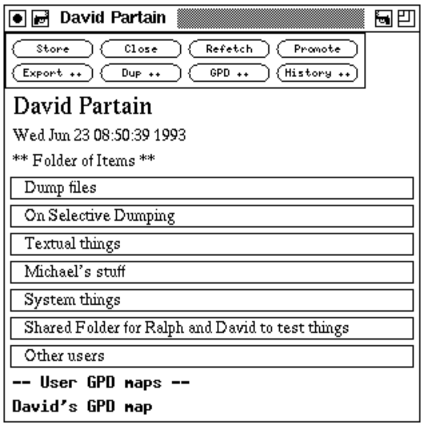

Figure 7.2: View built using 'user' GPD

The notion of a GPD was introduced as a way of separating the form of a view (or document presentation) from the contents. Ideally, the same contents can appear in several different views in different ways, depending on the particular purpose for the view. As always, the intention is to make it simpler for a user to see the information that is particular to his needs.

The idea of using GPD's may be illustrated in a hospital environment, where e.g. a patient record might (or should) look different for the physician and for the nurse. The nurse is involved in everyday care and should therefore be presented with the essentials for that task, while the physician may need an overview record relating the particular patient with other patients having the same symptoms or sickness. However, the patient is the same for both the doctor and the nurse, and the source of the information is the same. It is only the choice of which parts of that information to view which is different. The GPD is the mechanism used to select the information to view.

7.2 GPD Language

A GPD consists of a collection of declarations divided into five sections: STRUCTURE, ACCESS, FORMAT, EXPAND, and AUTOLINKS. Each of these sections is in turn divided into direct and indirect references; direct references are defined in that GPD and indirect references are defined in another GPD. In the subsequent sections, we will go into more detail about each of the GPD sections and give examples of how to use them.

All declarations in all sections are concerned with information that should be tied to the infonodes within the reference structure (the internal representation in xlincks). The kind of information to be bound to that keyword is dependent upon which section of the GPD it is in. Each reference in the section, whether an indirect or direct reference, is a sequence of "keyword 'values to keywords' ". For example, such a declaration might be

paragraph

LINK TEXT Paragraph

where ‘paragraph’ is the keyword and ‘LINK TEXT Paragraph’ is the value assigned it. (In this case, this declaration is in the ACCESS section.) The ‘paragraph’ keyword is regarded as a label for a concept that occurs within the type of object being defined in the GPD. For example, such a keyword may be ‘paragraph’, ‘title’, ‘correspondent’, ‘department head’, ‘author’, and so on, where the keyword is a conceptual part of the objects for which the GPD is being written. In the ‘text’ GPD, for example, ‘text’ is defined as having a singular ‘title’, a plural ‘paragraph’, and a plural ‘subtext’ decomposition. These are the keywords for this particularly GPD, which can be seen in figure 7.3.[1]

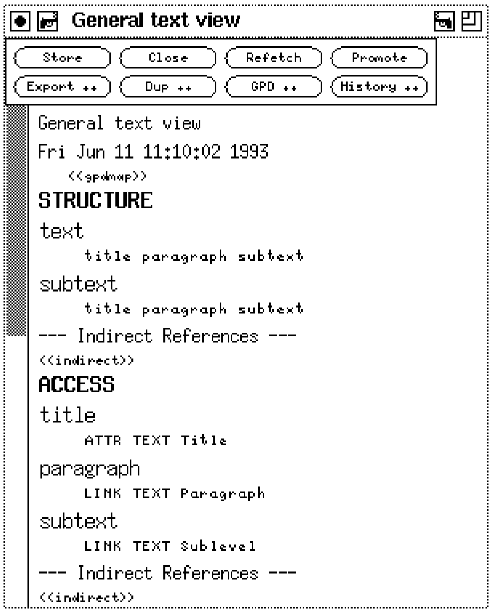

Figure 7.3: The ‘text’ GPD

Each section in the GPD has a particular function.

- STRUCTURE: View decomposition is done in the STRUCTURE declarations. This means that you, as the GPD writer, must break the view down into its subparts. Only you can know how far this decomposition needs to go. If a view subpart has logical subdivisions, then it should be decomposed. Figures 7.4 and 7.5 show the STRUCTURE sections of two GPD’s.[2]

[1] Not all of the GPD is visible in the figure, but rather only those parts which are necessary for understanding the illustration.

[2] Throughout this chapter, we will try to include many examples of that which we are discussing. Of course, we expect that you will also look at examples in the database.

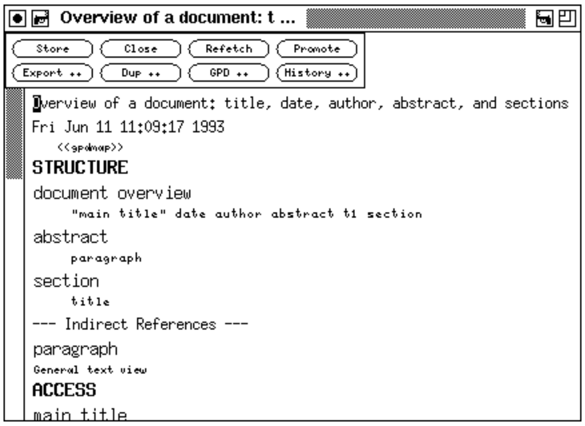

Figure 7.4: STRUCTURE of 'document overview' GPD

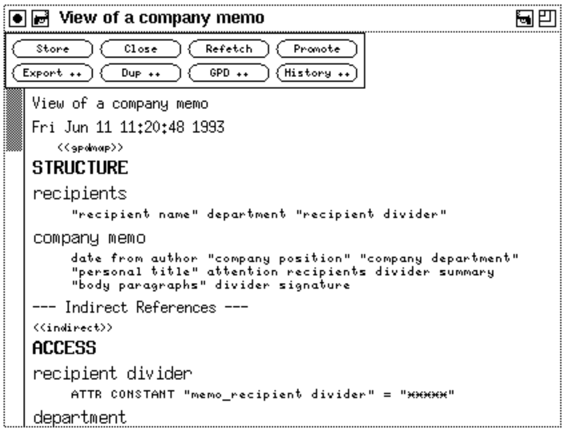

Figure 7.5: STRUCTURE of 'company menu' GPD

- ACCESS: Singularity versus plurality of a view subpart is declared in the ACCESS declaration. This means simply whether there can be on or more of such a subpart. The most common plural access declaration might look like:

  paragraph
  LINK TEXT Paragraph

  which means all of the linkitems in the link group TEXT:Paragraph. A common singular item is an attribute value, such as:

  title
  ATTR TEXT Title

  Secondly, the ACCESS section tells where to retrieve a view subpart within database objects. This will be explained in section 7.2.2. Figures 7.6 and 7.7 show examples of the ACCESS section.

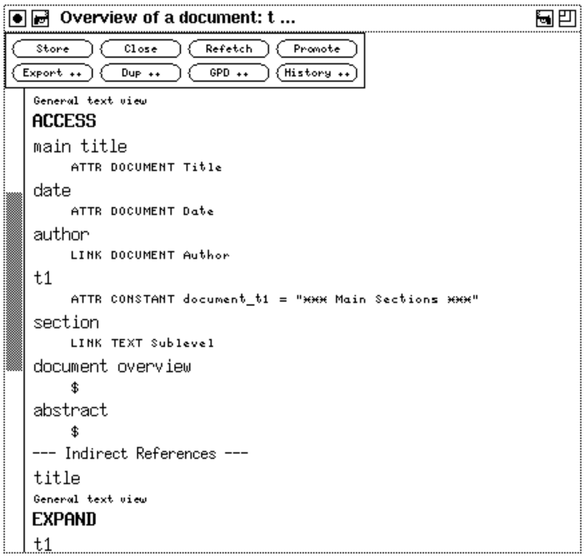

Figure 7.6: ACCESS of 'document overview' GPD

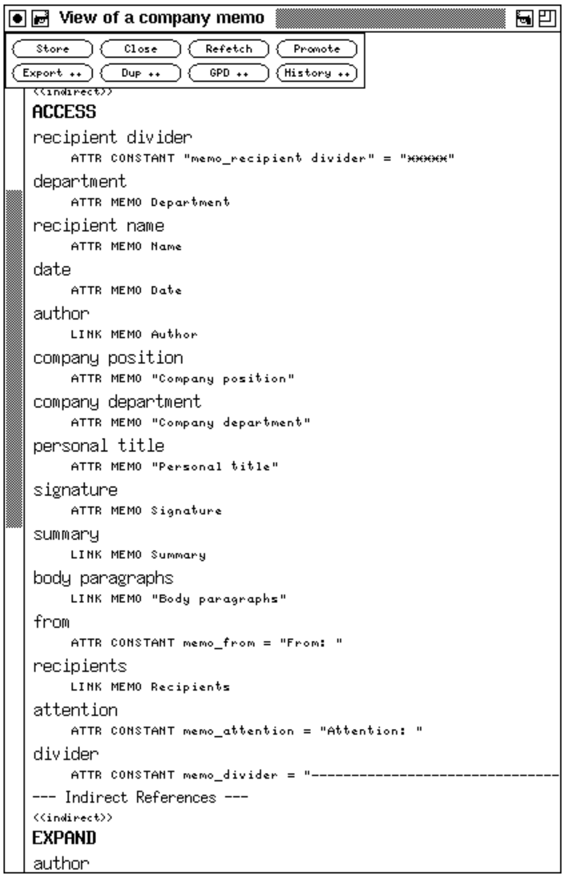

Figure 7.7: ACCESS of 'company menu' GPD

- EXPAND: Declares which GPD should be used if a subpart of that view is expanded. From figures 7.8 and 7.9, we see that in the 'document overview' GPD, expanding upon the main title will expand the object in 'full document' view. We also see that expanding the author item in the 'company memo' will expand that object in the 'user' view.

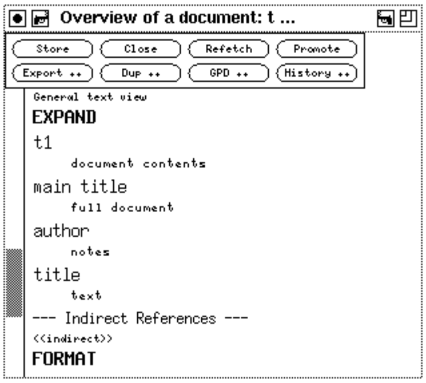

Figure 7.8: EXPAND of 'document overview' GPD

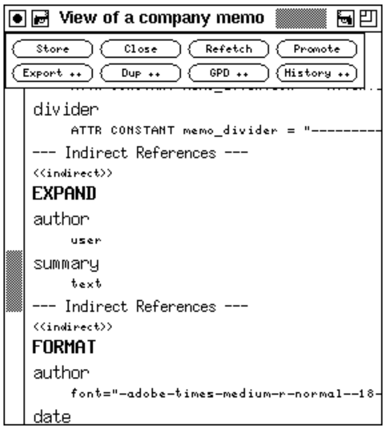

Figure 7.9: EXPAND of 'company menu' GPD

- FORMAT: Defines generic formatting attributes for how the subparts should be displayed. Several examples will be given in section 7.2.4.

- AUTOLINK: Provides an advanced feature allowing component objects in a composition to inherit links at creation. This will be explained below. Examples will be given in section 7.2.5.

7.2.1 STRUCTURE

Again, the STRUCTURE section of a GPD declares the structural decomposition of a view. Beginning with the most all-inclusive concept, — that for which the GPD is being defined, e.g. company memo, glossary, full document, etc. — you break down the view into its logical subparts. These subparts may then need to be further broken down. That is, the STRUCTURE section tells whether a view is made up of subparts, and, if so, what those subparts are. The most elementary subparts is called 'item' and is defined globally.

Again, figures 7.4 and 7.5 show examples of this.

Note that the most abstract concept (in the case of figure 7.4, this would be 'document overview') is the name of the GPD and must also be the name in the GPD map of which the GPD is a member.

Figure 7.10 shows the STRUCTURE section of the 'full document' GPD. From this, we see that a composition of type 'full document' is made up of the subconcepts "main title", date, author, abstract, and subtext. (Note that "main title" is in quotes because there is a space in the name. Otherwise it would be treated as two subparts.) This is all of the STRUCTURE defined in this GPD. Additionally, the STRUCTURE of the subtext is inherited the 'text' GPD (this is actually a *link* to that GPD) and the STRUCTURE of the abstract is inherited from the 'document overview' GPD. One could expand on 'General text view' in order to see the GPD from which subtext's STRUCTURE is inherited. Note that when you inherit the STRUCTURE of an object, you also inherit its ACCESS, FORMAT, and AUTOLINKS. The inverse is not true, however.

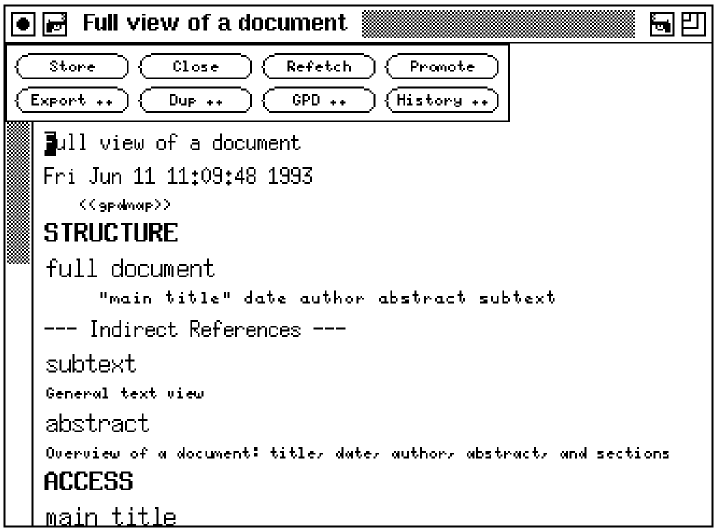

Figure 7.10: STRUCTURE of 'full document' GPD

Which parts of the STRUCTURE section will actually be visible? If you think of the STRUCTURE section as building a tree structure, then the leaf nodes will be those visible on the screen. That is, those subparts of the STRUCTURE which are not broken down into subelements will be viewed. Therefore, given the GPD in figure 7.10, we see that the main title, date, author, subtext titles, and subtext paragraphs will be visible. sideways tree structure shown in figure 7.11 shows the internal *xlincks* representation of this tree structure, called the 'reference structure'.

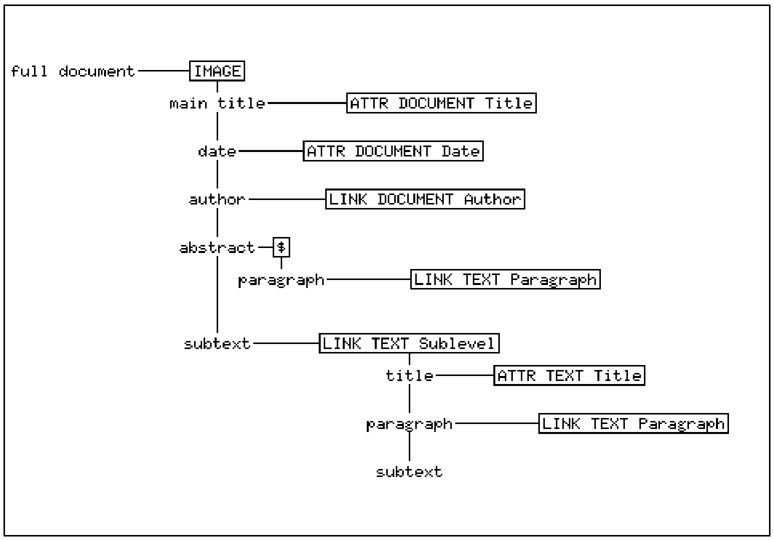

Figure 7.11: The reference structure

Therefore, the first step in writing a GPD is to decide what sort of structure you wish to have and to define locally what you cannot inherit from other GPD's.

7.2.2 ACCESS

The ACCESS section of a GPD declares how the concept values are associated with the underlying database, i.e. which "physical" database object a "logical" concept refers to and where to find it in the database — the path to the parts of the view. Therefore, for each subpart listed in STRUCTURE section, you declare how to find that part in the database.

In order to make it easier to understand the examples in this section, we now introduce a set of GPD language keywords and some specialized syntax, an understanding of which will help in following the discussion.

- ATTR: keyword for ACCESS'ing attribute values.
- LINK: keyword for ACCESS'ing link items.
- IMAGE: keyword for ACCESS'ing the image of an object.
- "%GROUP%": this macro means to substitute the group tag of the parent into this part of the ACCESS declaration.
- "%VALUE%": this macro means to substitute the value of the parent into this part of the ACCESS declaration. The value of the parent is dependent upon the its type but is generally that part which would be displayed on the screen if the parent is shown.
- $: this is the default ACCESS value, which is "same-as-parent". If no other ACCESS is given for a subpart listed in the STRUCTURE section, $ will be used. You do not need to include it, but may if you wish.
- ATTR CONSTANT Name = "initialize attribute to this value": This is both a way of declaring ACCESS as well as initializing a value. If the attribute CONSTANT:Name (with the names you choose, of course) exists, its value is used. If, however, that attribute doesn't exist, it will be created and initialized to the string "initialize attribute to this value".

In the following section, we will discuss the kinds of access that can be made to database objects and give examples. Please refer to these examples as the discussion progresses.

Keep in mind that each information component is made up of attributes, links and an image. Figure 7.12 shows such an information component.

Attributes and links are named using the group name, the field name, and for links, the position. For example, a link might be named by TEXT:Paragraph:1, meaning group TEXT, field Paragraph, position 1 (positions begin at 1). An attribute value might look like TEXT:Title, meaning the TEXT attribute group and Title field under that group. There can only be one attribute of this name in a given information component.

Presently, one can refer to the following database concepts:

1. All attribute groups in an information component

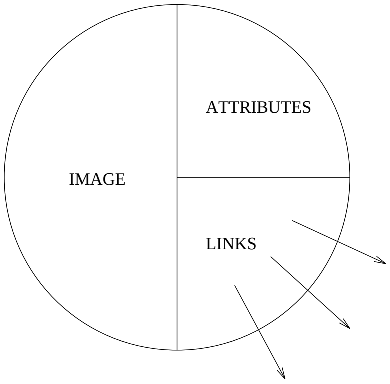

Figure 7.12: Parts of an 'information component'

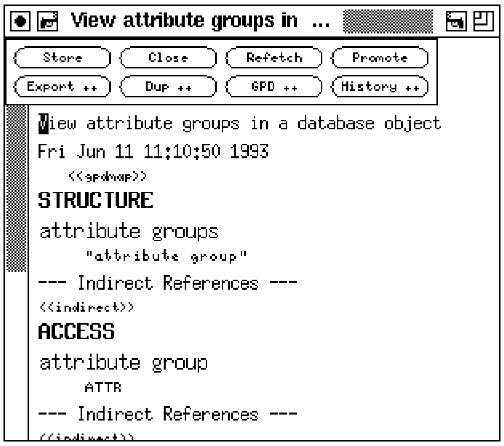

Figure 7.13: ACCESS of all attribute groups

For example, the example in figure 7.13 would retrieve all attribute groups and display the attribute group names.

Again, ATTR is a keyword.

2. All fields within a certain attribute group in an object

   - From figure 7.14, we see that the attribute groups are first accessed using the ATTR keyword. Thereafter, each attribute group is searched for its field names using the line 'ATTR "%VALUE%"'. %VALUE% means that the value of the parent (in this case, whatever attribute group we're looking through) will be used as the attribute group name in accessing the fields. For example, if the object has two attribute groups, SYSTEM and TEXT, these will be accessed by the "ATTR" access line. Then, using each of these, "SYSTEM" and "TEXT" will be used as access for attribute fields – ATTR SYSTEM and ATTR TEXT respectively. This will in turn display the group and field names. The output might look like figure 7.15.
   - If you only want to see the field names for the attribute groups, you would remove the "tag" from the attribute groups STRUCTURE definition. Keep in mind that if no access is defined (as for "tag"), then the default access is "same-as-parent". What this means, in this case, is that whatever would be visible in

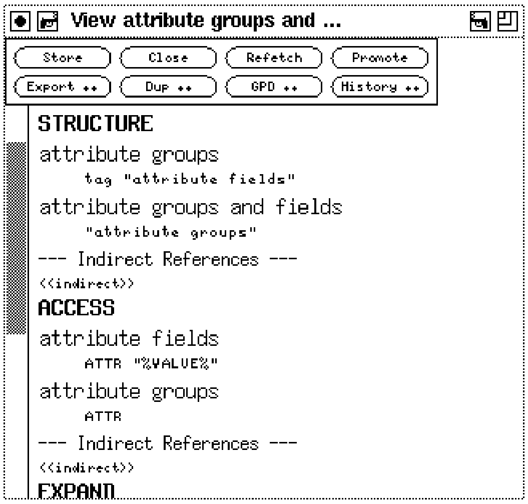

Figure 7.14: ACCESS of attribute fields

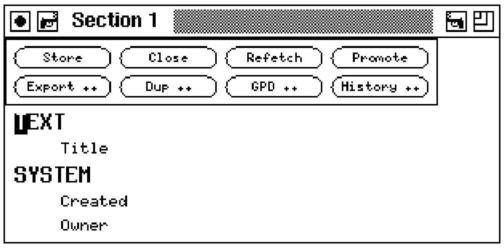

Figure 7.15: Output using previous GPD

the parent's node (the "value" of that node) will be displayed wherever "tag" is to be displayed. Normally these parent nodes, which we often call inner nodes, are invisible and only their substructure is visible. Same-as-parent, however, provides a mechanism whereby that inner structure can also be seen. Using the same object as the root as we used in figure 7.15, removing 'tag' from the STRUCTURE line for "attribute groups" would give you a view such as figure 7.16.

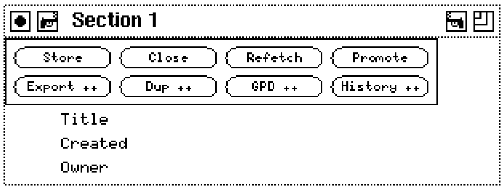

Figure 7.16: View without 'tag'

- If, rather than displaying all groups and fields, you wish to display a specific group and its fields, it becomes a bit more difficult. There are no uneditable constants in the *xlincks* system, meaning that everything you see is a part of the database and can be edited. Consequently, anything that you want to display must be in the database. For example, if you want to display just the string "SYSTEM", it must be in the database. (By convention, we put such things into the attribute group CONSTANT.)

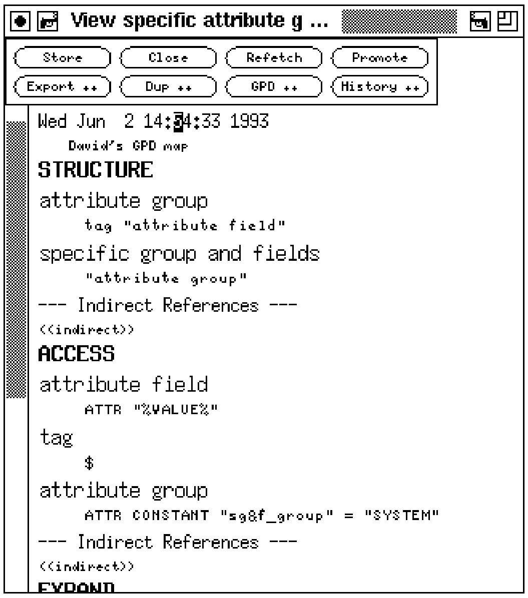

Figure 7.17: ACCESS of specific attribute fields in a group

Given that, look at the GPD in figure 7.17. It is used to view a specific attribute group. Firstly, the topmost concept, 'attribute group' is defined as a constant in the ACCESS section. That attribute value (the constant) is used as the access to retrieve the fields in that attribute group. This is because that concept is the parent that will be used for the resolution of the parent VALUE in *ATTR "%VALUE%"*. The "tag" field is "same-as-parent" (the $ sign), which means that it displays the value of the parent, in this case the attribute value **CONSTANT:sg&f_group**. If you then edit the attribute value in a view using this GPD, the fields in the attribute group that you change it to will then be displayed. Again, applying this GPD to the same object in the two previous figures, you see the initial view in figure 7.18. After then editing SYSTEM and changing it to TEXT (which alters the value of the attribute), we obtain the view in figure 7.19. This is a rather difficult example, so don't be too concerned if you don't understand it when you first begin using the system.

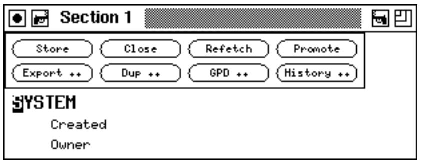

Figure 7.18: Initial view of SYSTEM attributes

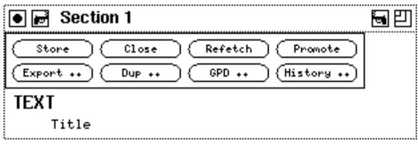

Figure 7.19: View after changing SYSTEM to TEXT

3. The value of a certain field within a certain attribute group

   In the example in figure 7.20, we again see that all attribute groups are retrieved using ATTR. The fields are then retrieved using the value of the parent (the attribute group name) as ATTR "%VALUE%". For each field, the value is retrieved using the group name in the parent ("%GROUP%") and the value of the parent (the field name). Thus, if there is a group SYSTEM with a field Created, the access for the value will be ATTR SYSTEM Created.

4. All link groups in an object

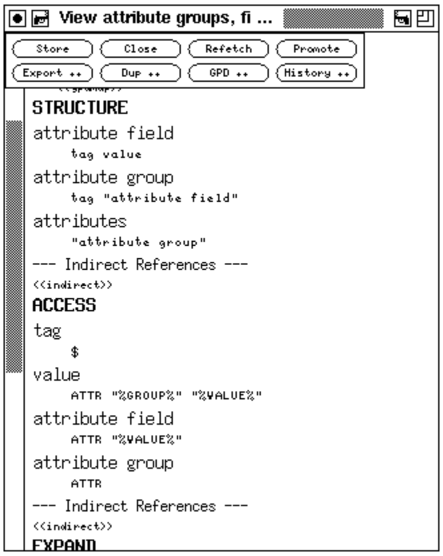

Figure 7.20: ACCESS of attribute values

This functions in the same way as showing all attribute groups in an object, but rather than the keyword 'ATTR', you use the keyword 'LINK'.

5. All fields within a certain link group in an object

   Again, this is the same as for attribute fields but with the keyword LINK rather than ATTR.

6. All items for a field within a certain link group in an object

   As with the previous two items, this works in a similar fashion as all the fields in an attribute group, but using LINK instead of ATTR.

7. The first item for a field with a certain link group in an object

   This is a special case which is seldomly used. Instead of the ACCESS directive 'LINK GROUP Name' (which would pick up all the link items in this link group), you use 'LINK GROUP Name 1', which specifies that you use the first link item in that link group. No other integer other than 1 is acceptable.

   This is available simply so that you can view singular link items even if there are multiple link items in that link group.

8. The image of an object

   This is accessed using the keyword IMAGE. Figure 7.21 shows the 'image' GPD, which does this.

Notes about the ACCESS section

Same-as-parent semantics: If a subpart has an ACCESS of same-as-parent ($), the value of the parent will be used for that subpart. Same-as-parent is the default access declaration if no access is given.

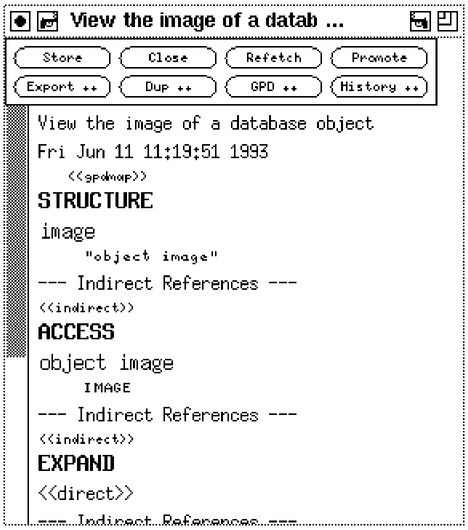

Figure 7.21: The 'image' GPD

Note that further expansion of a link item applies to the object to which there is a link rather than to the object holding the link. In this way, a larger database structure can be traversed.

7.2.3 EXPAND

The EXPAND section of a GPD is used to declare the default view (GPD) which should be used when an item is expanded from this view. For example, assume that you are viewing an object in the 'flat text' view, which consists of a series of paragraphs. In the 'flat text' GPD, you could declare in the EXPAND section that paragraphs should in turn be expanded in the 'notes' view, as in figure 7.22.

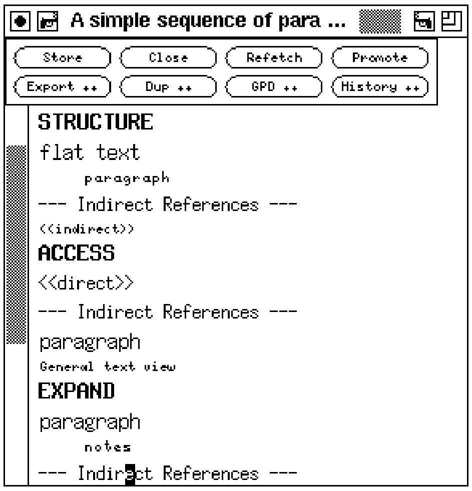

Figure 7.22: The 'flat text' GPD

Assuming that you are expanding an object of type 'paragraph', determining which view to use when expanding that object is carried out as follows:

1. First, the object being expanded (the "expansion object") is checked for the attribute EXPAND AS:paragraph and uses the GPD named there. Such an attribute might look like figure 7.23, in which case the 'paragraph's would be expanded as 'notes'.

Figure 7.23: 'EXPAND AS' attribute group example 1

2. Secondly, the composition root (the root of the composite object) is checked for the attribute EXPAND PART:paragraph and the GPD named there is used (see figure 7.24).

Figure 7.24: The 'EXPAND PART' attribute group

3. Thirdly, the GPD object itself is checked for the attribute EXPAND:paragraph, and the GPD named there is used.

4. Fourth, the expansion "EXPAND AS" attributes are checked in the expansion object. If the values for all the fields are the same, then that value is used for the expansion.

   For example, if the "EXPAND AS" attributes looked like figure 7.25, and you are expanding a paragraph which has not found an expansion GPD yet, 'notes' will be used as all other "EXPAND AS" values use 'notes'.

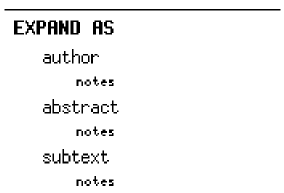

Figure 7.25: 'EXPAND AS' attribute group example 2

5. Fifthly, the tag (in this case 'paragraph') is assumed to be the name of a GPD.

7.2.4 FORMAT

The format part of a GPD declares formatting attributes associated with concepts. Formatting attributes take the form of X windows resource declarations. Some of the most common are: "font=somefont" (to set the font for that concept), "borderWidth=1" (to set a border around some concept), "width=300" (to set the width of that item in the window), "leftMargin=35" (to indent just a bit), and so on. Figure 7.26 shows the FORMAT section of the 'text' GPD, which includes some interesting formatting.

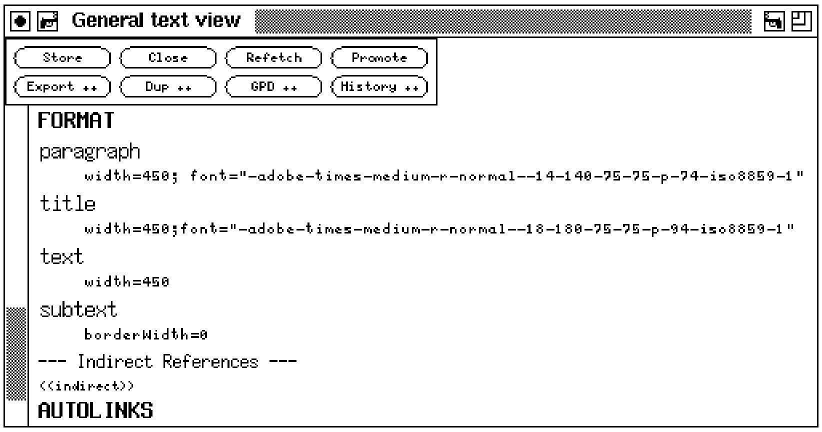

Figure 7.26: FORMAT in 'text' GPD

7.2.5 AUTOLINKS

The autolinking feature allows links to be given to new nodes as they are created within compositions. In a practical sense, this means that when an item is inserted in a composition by means of the M-L-I command, the new item node is assigned a link to something reachable from the composition root node. There are plans to extend this capability so that autolinking can be done to other areas of the database as well, but at present one can only autolink to objects available through the composition root.

How To Setup An Autolink

Autolinking is set up by adding information in a GPD describing the particular linking for an item tag (concept) when new items are created of that type. For instance, if an autolink is defined for 'paragraph' in the 'text' GPD, then each new paragraph inserted within a text view is automatically assigned the specified link at creation. In this way components can inherit information from the context in which they are created.

Let us say that new paragraphs in a text should inherit a link from the root of the text. The link to inherit might be the link TEXTROOT:Inherit:5 from the root of the text, and it should be called TEXT:Given:1 in newly created paragraphs. Then you would write the following AUTOLINK directive as a value for 'paragraph' in the AUTOLINKS section of the text GPD:

paragraph
    TEXT Given = TEXTROOT Inherit 5

Another example is the 'folder' GPD, which defines an autolink for 'item', so that each new item is assigned a back-link to the folder in which it is created. See figure 7.27.

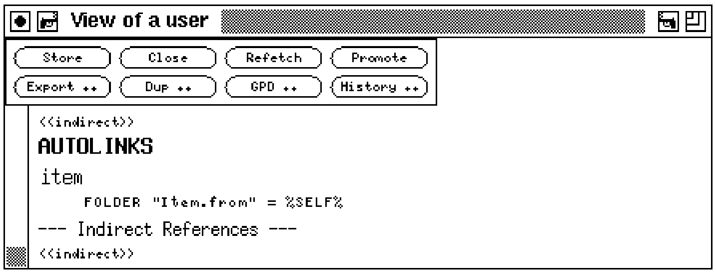

Figure 7.27: AUTOLINKS in 'user' GPD

Autolinking Syntax

Each autolink is defined by specifying the name under which the link should be put in the new object (here called 'new object') , and the path to the object to which the link should go (here called 'link object').

The root node of the composition in which the new item being created is going to be included can be referred to as '%SELF%'.

For example, the folder GPD contains an autolink for 'item' with the following specification:

FOLDER "Item.from" = %SELF%

Therefore each new 'item' created within a folder will get a link with group tag FOLDER and field tag "Item.from" (excluding the double-quotes) to the folder object. This link is set in an item object only when it is created within the folder and not, for instance, when it is moved from another folder.

Additionally, any object reachable from the root by following links can be specified as a link object. The specification is a chain of links separated by "+". Each link in the given chain is either:

1. a group tag, field tag, and a position index,
2. a group tag and field tag, with a default index of 1,
3. a group tag, using the field tag on the left side of the "=" and an index of 1, or
4. nothing after the "=", in which the group and field tags are taken from the left side and the index is the default of 1.

If you have multiple autolinking directives for one object, they are separated by a ';' (semicolon).

We will now give examples of each of these:

```
folder
          BACKLINKS Author = INHERIT Author  2
```

This autolink would create in every new folder a link called "BACKLINKS Author" and link in INHERIT:Author:2 as found from the composition root.

```
folder
         BACKLINKS Author = INHERIT Author
```

would behave in the same way, but would link to INHERIT:Author:1, since 1 is the default index.

```
folder
         BACKLINKS Author  = INHERIT
```

is exactly equivalent to the predecessor, since Author will be taken from the left side.

```
folder
         BACKLINKS Author  =
```

will create a link called BACKLINKS Author to the object found in the link BACKLINKS:Author:1 in the composition root. Both BACKLINKS and Author are taken as defaults.

```
folder
         BACKLINKS Author = INHERIT Author 2 + CHILD Tom
```

will create a link called BACKLINKS Author to the object found by following the second link in the INHERIT Author link group and following the link CHILD Tom 1 from that object. You can have as many "+" chains as you want, but you must specify the group and field tags for all of the chains. You can leave the last chain empty (between the last "+" and the ";") in which case the last group and field tag names will be used.

Figures 7.28, 7.29, and 7.30 show autolinking examples from GPD's on the system.

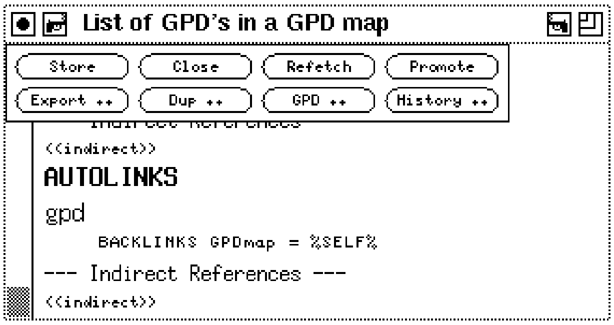

Figure 7.28: AUTOLINKS in 'gpdmap' GPD

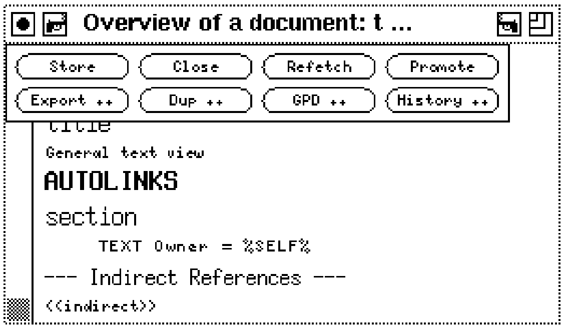

Figure 7.29: AUTOLINKS in 'document overview' GPD

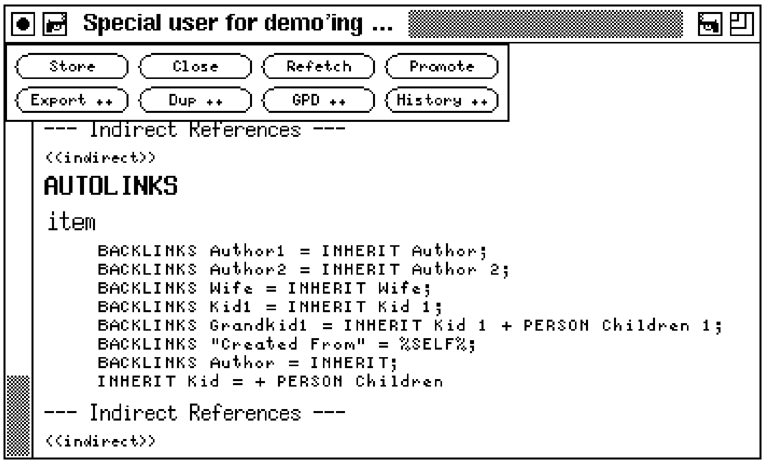

Figure 7.30: AUTOLINKS from a demo GPD

7.3 GPD Maps

The GPDs used by the system are found through a chain of GPD maps formed from the linkitems of SYSTEM:GPDmaplist in the user and the system root objects. The search for a GPD tag thus begins in the map SYSTEM:GPDmaplist:1 of the user root object, and thereafter the maps :2, :3, etc. are searched in order. If the tag is not found there, the search is repeated for the system root object. Each GPD map looks something like figure 7.31.

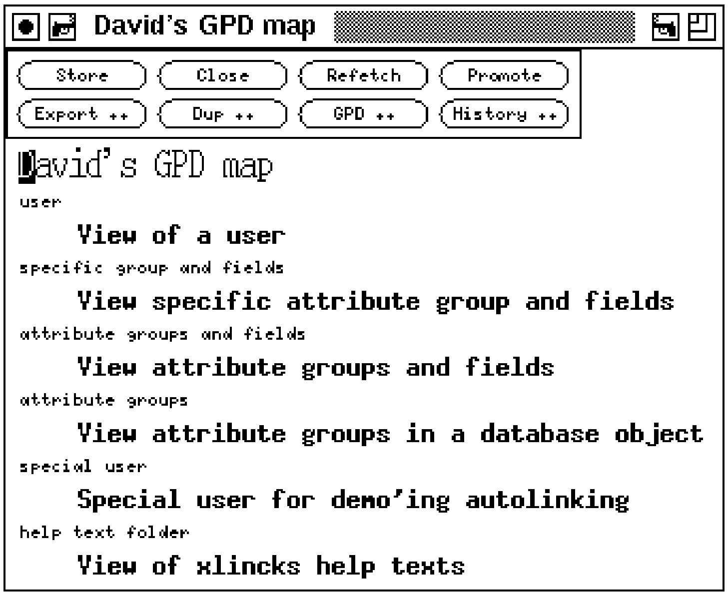

Figure 7.31: A GPD map

As you see, the GPD map holds a table that defines tags for GPD objects and a brief description of the GPD. It is this tag which must be the same as the topmost concept in the STRUCTURE section. The table is contained under the link group GPDmap, where each field defines the association between a tag (the field tag) and a GPD (the object linked to at position 1).

7.4 Tag Inheritance Chains

As previously touched upon, the declarations in GPD's may be linked together so that one GPD reuses some portion of another GPD. This is accomplished by a kind of inheritance policy, whereby a tag may be resolved by following a link rather than examining an attribute value. The tag value is then looked up in the GPD linked to. The declarations to carry this out are in the 'indirect references' section of each section of the GPD. Each declaration consists of the name of the tag to inherit and a link to the GPD from which the tag is being inherited. Note that this MUST be a link, not simply the name of the GPD.

If the STRUCTURE declaration is indirect in this way, then the subsequent definitions relating to the indirect tag are completely resolved using the indirection. That is, if the inheritance is in the STRUCTURE section, then all other sections (ACCESS, EXPAND, FORMAT, AUTOLINKS) are also inherited. However, if the inheritance is in the ACCESS, FORMAT, AUTOLINKS, or EXPAND declarations, the inheritance is valid only for that section.

As an example, compare the 'document overview' GPD declarations of paragraph and title in figure 7.32.

As you see, the 'paragraph' concept inherits its STRUCTURE from the text GPD, and therefore it has the same 'behaviour' whether in the 'document overview' or 'text' view. The 'title' concept inherits only the ACCESS declaration from the 'text' GPD, and therefore its STRUCTURE, FORMAT, and EXPAND declarations are taken from the 'document overview' GPD rather than the 'text' GPD. One result of this is that in the 'document overview' view expanding on the title will expand the corresponding section, because the EXPAND declaration for 'title' states that a 'title' should be expanded as 'text'. On the other hand, if 'title' inheritance were done in the STRUCTURE section, the EXPAND section would be taken from the 'text' GPD, which states that a 'title' should be expanded in 'document contents' view. Note that if you wish to see what is being inherited, you can expand on the name of the GPD in the 'indirect references' section after the name of the concept being inherited.

7.5 System Defined GPD's

The system level GPDs are divided into six GPD maps:

- GPD's for view GPDs and GPDmaps: views for GPD's and maps of GPD's.
- Text GPD map: several views over basic text and document structures.
- Folder GPD map: views around a basic folder structure.
- System GPD map: certain system related views.

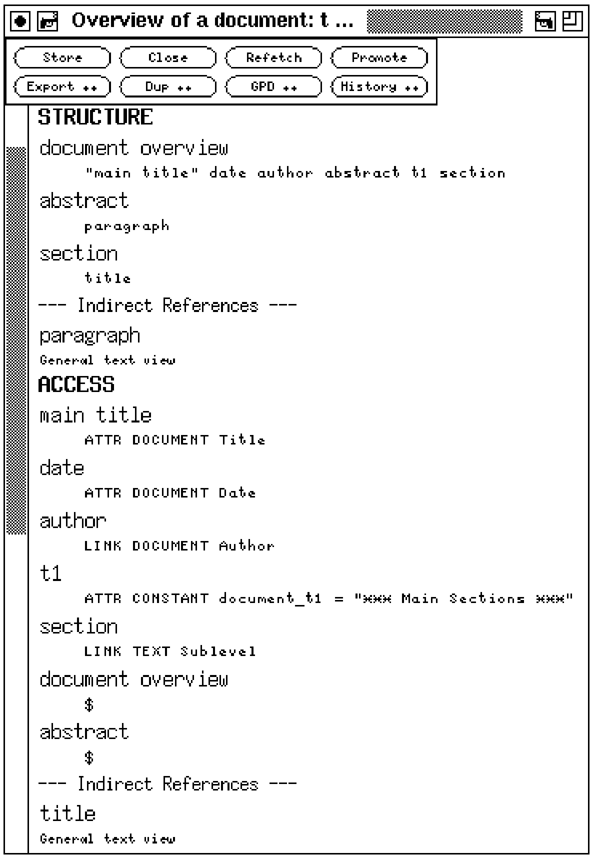

Figure 7.32: The 'document overview' GPD

- Map for viewing parts of database objects: for view whole nodes, the image, the links, or the attributes of a component object.
- Dump GPD map: views regarding the selective dump facility.

All of these GPD's are available on the system and as such will not be reproduced here.

7.5.1 GPD's for viewing GPD's and GPDmaps

This GPDmap has just what the title says.

The 'gpdmaps & contents' view

The 'gpdmaps & contents' view applies to user root objects, and presents the user level GPD map in detail. It presents each map, all of the GPD's in each map, and the brief synopsis of each GPD.

The 'gpdmaps' view

The 'gpdmaps' view applies to user root objects, and presents a list of the user level GPD maps.

The 'gpdmap' view

The gpdmap view is used for GPD maps. This is used to view a single GPD map.

The 'gpd' view

The 'gpd' view is used for viewing GPD's.

7.5.2 The Text GPD map

The text GPD map defines the following views, all of which have to do with text processing of one sort or another:

document overview, full document, text, notes, document contents, flat text, glossary, glossary index, and company memo.

The 'document overview' view

The 'document overview' view presents an overview of a document including the main title, the authors, the date, the abstract, and the major section headers.

The main title expands into 'full document' view by default, the authors expand into 'notes' view by default, the sections header expands into 'document contents' view by default, and each section header expands into the section in 'text' view by default.

The 'full document' view

The 'full document' view presents a full document view including both the overview information and the full contents of all sections.

The main title expands into 'document overview' view by default, and the authors expand into 'notes' view by default.

The 'text' view

The 'text' view defines the general storage of text in a recursive structure of paragraphs and subtexts. The 'text' view itself is a view which is intended to be included in other views where texts are desired.

The ‘text’ view interacts with the ‘title’ and ‘notes’ view so that expanding upon a section title yields a ‘title’ view of the section, and expanding upon a paragraph yields a ‘notes’ view of the paragraph.

The ‘notes’ view

The ‘notes’ view presents text as annotation for an object image.

The ‘document contents’ view

This view presents a full list of contents for a document, including the main title.

The ‘flat text’ view

This view presents the list of paragraphs in a text section, without either header and inner subsections.

The ‘glossary’ view

This view is a one-level text view intended for glossaries.

The ‘glossary index’ view

This view is a one-level list of contents view of text. Each entry expands in its definition.

The ‘company memo’ view

The ‘company memo’ view provides a template for a memo. It has a relatively strict layout.

7.5.3 Folder GPD map

The Folder GPD map contains views around a basic general document structuring system.

The ‘folder’ view

This view presents an operational view of a folder. This includes the name of the folder, its date of creation, and all of the items in that folder.

The ‘folder & default expand type’ view

This view presents a detailed view of a folder where the specific item expansion views are included and can therefore be altered. This is the view that is used if you wish to set the default expansion for all items in a folder.

The ‘item’ view

The ‘item’ view is a focussed view on the item expansion view attribute for the selected item. The view is applied as the last resort upon expanding an item, and it allows the specific view for that item to be filled in (for subsequent expansions).

7.5.4 System GPD map

The system GPD map contains important auxilliary GPDs. Normally only the ‘user’ tag is overridden by a declaration in the user GPD map.

The ‘user’ view

User root objects are normally expanded in the ‘user’ view, which presents the user root object as a folder followed by the user level GPD maps.

The ‘define expand types’ view

This view applies to any object for filling in specific context-dependent default expansions on the object. That is, this view can be used to set how to expand an object in a particular context.

The ‘define SPD’ view

This view applies to any composition root object for filling in specific presentation information.

See the SPD document in the Help! window for more information about SPD’s.

The 'login profile' view

This view applies specifically to the user root object for filling in which views should be brought up directly at login.

7.5.5 Map for viewing parts of database objects

Each of these GPD's shows a particular part or parts of a database object. They are particularly interesting in that they show the exact makeup of a database object. If you are uncertain what part of the object is being shown, these GPD's will allow you to discern that.

The 'node' view

The 'node' view applies to any object and it presents the complete NODE information for the object (attributes, links, and image).

The 'image' view

View the image of the object.

The 'links' view

View all of the link groups with their fields and link values.

The 'attributes' view

Show all of the attributes groups with their fields and attribute values.

7.5.6 Dump GPD map

The Dump GPD map contains GPDs related to the selective dump facility described in chapter 6, which enables you to dump selected parts of a database to a t2lincks file for inclusion in a separate database.

The 'dump file description' view

View used to create and view the objects which define a particular dump of a part of the database.

The 'dump file folder' view

Chapter 8

Concept Dictionary/Glossary

- Auxiliary windows: there are two kinds of windows in the *xlincks* system which don't directly display information from the database. These are history graphs and the reference structure graphs.
- Binding Table: tells the versions of the components which make up a version of a composite object.
- Command History: a record of the relative order in which database and workspace operations have occurred and which object versions were affected. It is intended for use in future A.I. reasoning and for construction of past states of the database.
- Command Menu: the first window which appears when you start *xlincks*. It contains buttons for executing various *xlincks* commands.
- Composite object: an object which is made up of a root object and one or several 'components' (see below). A composite object is defined by combining a root object and a Generic Presentation Descriptor. A composite object is displayed in a single window.
- Component: a part of a composite object. Each component is a single information component. Note that it is possible that only a particular part of the information component might be visible in the window as an 'item'.
- Expand: in the context of *xlincks*, expansion means that a composite object item is 'opened' for viewing and editing. When you expand on an item in a window, you open a window holding a composite object where the root of the composite object is the component to which that item belongs.
- Generic Presentation Descriptor (GPD): the template (form) used to create a view (see below). The combination of a root object and a GPD is used to create a composite object. The GPD defines the possible components of a composite object, how to access them in the database, which parts of the components show be displayed in a view of the composite object, as well as any special formatting characteristics,
- GPD map: a group of GPD's which may or may not be related in some way. Simply a grouping mechanism. The **LINCKS** system looks for GPD's in the GPD maps in the system root object and in the user root object. Other maps (either new ones or other users' maps) that you wish to use will have to be linked in to the user root GPD map.

- History: for the user of *xlincks*, history usually means temporal history — or the history of how an object (either a composite object or a information component) has developed over time. See also 'Command history' above.

- Information component: a single database unit made up of links, attributes, and image. An item in the window is a displayed part of an information component. An information component can be viewed using the 'node' view.

- Item: the smallest indivisible unit in a view, for example a paragraph, title, date, etc. Each item is an indivisible part of an information component (attribute, image, link tag, etc.).

- Parallel Editing: given the multi-user nature of the **LINCKS** system, and given the ability to share objects, it might be a common occurrence that two users edit a particular information component simultaneously. (Note, of course, that the item that they edit might not be the same part of the information component.) When this occurs, the users are warned that this is happening. At present, there is no automagic merging of the parallel objects into a single object. Rather, they will be parallel versions in the history and the last one saved will be the 'latest', which means that it will be the one retrieved by default the next time that object is retrieved. If merging is desired, it will have to be done manually.

- Placeholder: indicates that an item in that view is non-existent in the database (when surrounded by << ... >>) or that it is a null string (when preceded by *empty:*) Sections 5.3 and 5.4 give a full explanation of the semantics of both.

- Promote: promote means to copy an old version of either a composite object or a component and make that be the latest version of the object. In so doing, you essentially go back in time to use an older version. Without promoting it, an old version of an object is not editable.

- Reference structure: an internal structure used to represent the logical decomposition of a composite object as described by a GPD. The reference structure consists of a *template* structure, which is a direct representation of the GPD, and a *target* structure, which contains the actual information items obtained by applying the template (GPD) to the database relative to a given root object.

- Root object: an information component which is conceptually at the 'top' of the reference structure. It is the beginning of the 'search path' in building the view.

- Transient: a version of an object which has been created in a user's workspace but which has not been saved to the database. Transient objects disappear if they are not saved before logging out.

- View: a representation of a composite object on the screen. The difference between a 'composite object' and a 'view' is that a view is made up of 'items', not 'information components'.

Index