A Hypertext-Based Data Flow Diagram Editor: Experience in HyperCard™ Prototyping

Mel Hatzis and Jacob L. Cybulski
Amdahl Australian Intelligent Tools Program
Department of Computer Science and Computer Engineering
La Trobe University, Bundoora, Vic 3083, Australia
Phone: +613 479 1270, Fax: +613 470 4915

Abstract

This paper describes DFDEdit, a prototype graphic editor for the creation and modification of Data Flow Diagrams. The paper deals with the issues of the editor user interface, the representation of the diagram information contents, and the system ability to browse through a document repository in a hypertext-like fashion. A brief description of the prototype implementation in Apple Hypercard™ is also given.

1 Introduction

HyperCASE

The HyperCASE project brings together a complete suite of loosely coupled Computer Aided Software Engineering (CASE) tools. The tools provide the support to software developers in the planning, specification and design stages of the software life cycle, via both textual and diagrammatical presentation techniques embedded in a hypertext framework. The HyperCASE system allows its users to create software design documents (e.g., Data Flow, Entity-Relationship or Nassi-Schneidermann diagrams) which could subsequently be browsed using their contents rather than their physical organisation.

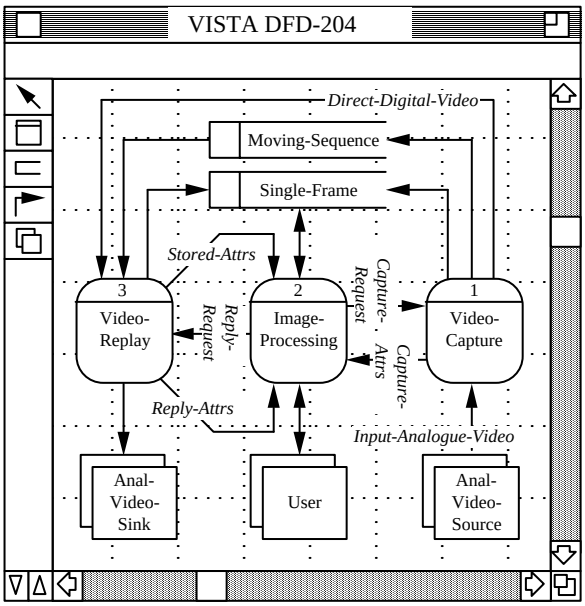


Figure 1 - Sample DFD

This paper describes a prototypic diagram editor developed to

a. gain experience in the design and implementation of editors of this kind for the development of a fully customisable diagram editor for HyperCASE,

b. develop techniques for allowing all components of a diagram to be treated as a button.

The prototype was developed using a commercially available hypertext system, namely Apple Macintosh Hypercard, which will be the hypertext system referred to in this paper.

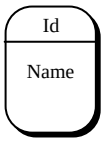
Data Flow Diagrams

Data flow diagrams are one of the most popular and effective representations for functional analysis and structured design.

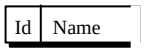
The main objective of data flow diagrams is to pictorially describe the flow of data in a software system using graphical objects representing the system processes, data stores, external entities and flows (see Fig. 1).

The DFD Edit diagram structure and the shapes of diagram components conform to the slightly modified Gane and Sarson notation, which is as follows:

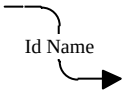
double squares denote *external entities*, i.e., sources or destinations of data external to the system, e.g. operators, clients, etc.;



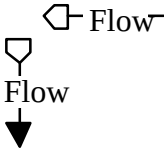
rounded rectangles denote *processes* transforming data flows, e.g. receipt of an invoice, posting a transaction to the ledger, or production of a report;



open-ended rectangles denote *data stores*, e.g. files or databases;



arrows represent *data flows* transformed by the system processes, e.g. a data entry form, a cheque requisition, request for information;



frequently a data flow diagram spreads over many pages and some of the flows have to cross the page boundaries, *terminators* are used to depict the continuation of flows over split pages.

Each of the DFD processes may be defined, or refined, according to the top-down analysis terminology, by either another diagram or by some kind of an activity chart (lowest level of definition, e.g. Nassi-Schneidermann charts, structured English or flowcharts). Data flows and stores are usually specified in a data dictionary notation or via Entity-Relationship charts.

The automation of DFD production relieves the system analysts from laborious drafting of processes, stores, entities and flows, alleviates the need for frequent redrawing of diagrams, allows for the delivery of professionally looking software designs. Additional tools for the integration of software documents into readily accessible repositories with the capability of inter-document navigation would certainly enhance the DFD editor capabilities.

The following sections examine the factors needed to be considered in developing a graphical editor for DFDs, the editor function, diagrams layout and their aesthetics.

Hypercard

Hypercard™, developed by Apple Computer and Claris, was selected as a prototypic platform under which the research described in this paper was conducted. It is therefore useful to describe some of the basic properties of Hypercard.

Hypercard supports the creation of the following types of components:

- *buttons* which can be clicked on by a mouse to perform certain activities, such as navigating to a different card, or performing some mathematical calculation(s), the button attributes include their style, size, position, highlighting and naming conventions, associated icons, etc.;
- *fields* which hold textual information, their attributes also include the style, size, and position, but also the text font, its size, style, and margins;
- *cards* on which the buttons and fields can be created and positioned;
- *backgrounds* which contain common characteristics for a collection of cards;
- *stacks* which are the card repositories;
- *scripts* are the HyperTalk code describing the behaviour of Hypercard components, they consist of modules termed *functions* and *handlers* monitoring events related to the components in which they reside (e.g. a key-stroke or a mouse-click).

The Hypercard features allow the creation of movable buttons representing the DFD processes, stores and entities, or to assume certain hypertext properties by DFD documents, like navigation links between the processes and their refinement diagrams.

One of the main requirements for HyperCASE is its ability to efficiently handle large numbers of documents and the links between them. Although Hypercard is generally adequate in producing small prototypes it does not meet the excessive requirements of the HyperCASE project which will handle documents in the order of 10^6 pages. In order to handle such a large number of inter-document links efficiently, the use of a database external to Hypercard is necessary.

2 Data Flow Diagram Editor (DFDEdit)

Layout

DFD Edit windows (see Fig. 1), similarly to other HyperCASE windows, consist of three partitions :- the tool palette, the work and the message areas allowing for the selection of an editor tool, the construction and manipulation of graphical objects with the selected tool, and the display of system messages, respectively.[1]

The message area is used to display information consisting of error and help messages which are useful in informing the designer about the state of the system. Some more intuitive errors result in a beep without a message being displayed.

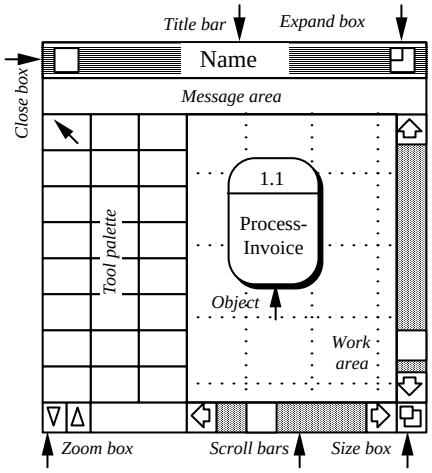


Figure 1 - DFDEdit window layout

The selection of either the *process*, *flow*, *data store* or *external entity* tool from the tool palette of the DFD editor, allows the designer to create and position one of the components in the work area. The *browse* tool, is used for viewing the diagram, and navigating to various pages making up the design document.[2]

Function

The tools selected from the palette trigger the editor to enter an operational mode which allows a diagram component to be created on the screen. The mode will then persist allowing the repetitive application of the same function by the designer to the diagram.

With a tool (other than a *flow*) chosen, the user may position the mouse cursor at a particular place in the work area and press the mouse button down to create a tool-specific DFD component. By the subsequent dragging of the mouse, the component

---

[1] Note that tool refers to a mechanism which is used to automatically generate some specific graphical feature in the diagram.

[2] A reasonably sized system of DFDs would take up more than one page (screen), and viewing its diagrams would require navigation from one portion (page) of the DFD to another.

can be re-positioned to a more desirable location, and then its text fields may be typed in as required (most likely the component id and its name).

When creating a data flow, the designer must click the mouse onto an existing non-flow component (i.e. external entity, process or a data store) to select a flow source. While the mouse is dragged over the work area the flow lines are drawn. Bends may be generated in the flows by altering the mouse movement direction between mouse clicks. The flow definition is completed when either the user clicks onto a target component (other than a flow) or double-clicks on the background thus declaring the flow to be continued elsewhere (for a visual feedback of this action a special terminator component will automatically be appended to the flow). The start of a flow and its end, symbolised by an arrow-head (except for the terminator), are automatically aligned to the edges of the source and target components, respectively.

Once a graphical presentation of a DFD component is accomplished (i.e. its shape, position, size, id and name), then a dialogue box containing the conceptual information about the component may be displayed and possibly altered (e.g. the name of the analyst defining the component, the programmer responsible for its future implementation, the allocation of component-related resources, etc.), thus completing a DFD component specification.

When finally completed, a set of DFD documents may be searched through by placing the editor into a browsing mode and the subsequent selection of individual diagram components with the mouse. Thence, clicking the mouse onto a process button will navigate the user to the process refinement document (whether another DFD or an activity chart), the selection of a flow segment or its name (or data store icon) will lead to a data dictionary document, the flow terminator will react by displaying a page with a flow continuation, its appropriate source or a destination.

Aesthetics

Some aspects of diagram aesthetics can be easily accommodated and fully automated in a computer-aided diagramming tool. Aesthetics alone may lead to a significant improvement in diagram clarity and a decrease in its complexity. It may reduce the time and the cost of producing charts and graphs, and could enable the creation of better-structured design documents. Overall it increases the maintainability of generated documentation and thus improves the productivity of designers.

The ability to aid the construction of well-structured data flow diagrams was seriously considered during the implementation of our DFD editor. We recommended the following diagram properties as especially important:

a. the global area of the diagram should be minimised;
b. the flows ought to consist of vertical and horizontal lines only;
c. the flows must not cross over any non-flow components;
d. the number of crossings between flows should be minimised;
e. the number of bends in the flows should be reduced.

Some of the recommendations (e.g. b and c) were readily incorporated into the DFD Edit, others (i.e. a, d, e) rely heavily on the use of an invisible grid[3] aiding DFD component alignment, preventing their overlaps and collisions, allowing a better utilisation of space available drawing, etc.

The icons depicting DFD processes, external entities and data stores are positioned, moved and re-sized within the work area in accordance with a widely spaced grid. Although, the primary purpose of using such a grid is to assist the designer in mutual alignment of the diagram components, it also provides a way of unique identification of DFD components within the work area by the grid coordinates. This facilitates the prevention of component overlapping or their collision during the user instigated manipulation.

The major difficulties in producing neatly laid out data flow diagrams arises, however, not in the positioning of iconic component of fixed shape and size, as is the case with DFD processes, stores and entities, but in the organisation of arbitrarily shaped, multi-segment flows. Some of the constraints put on the flow construction may be easily alleviated by also aligning them to the grid (b and e). The coarse grid used in positioning of non-flow DFD components, however, is not quite suitable for this purpose, as in some circumstances it may increase the global area of the diagram (due to the inability to draw lines close together) or may force certain flows to overlap (insufficient drawing space for diagram enlargement leading to the violation of a and c). Therefore, a thinner grid was chosen for the alignment of data flow elements.

---

[3] The grid technique is a widely used and described method of achieving neater layout

A problem with using a variable sized grid in editing different types of DFD components is that the arrowheads on the flows may not be positioned directly on the edge of the components they are linking. In order to overcome this, the size of the non-flow components is selected so as to force their perimeter to align with a thin grid line, even though its movement and manipulation is governed by the coarsely spaced grid. Since the flows are drawn according to the thin grid lines, the arrowheads are then positioned exactly on the component's perimeter.

3 Contents Database (CDB)

Component Information Contents

Every diagram constructed with a HyperCASE editor is represented in a Contents Database (CDB) for later reference by other HyperCASE tools. The representation ignores all of the component presentation attributes (e.g. shape, colour, position, etc.), which are graphics system dependent, and retains only the conceptual information about the components and their relationships (e.g. component id, its name, the creator and implementer information, its relationship with other components via attributes and flows). The components conceptual information may be normally viewed and altered through the information dialogue box being invoked during the components editing.

We now describe the specific use of the CDB in DFDEdit. The contents database contains two different types of information about DFD components. First is the component *definition*, e.g. a definition of a process as a DFD, Nassi-Schneidermann diagram, or C function, or the definition of a data flow using an Entity-Relationship diagram, a data dictionary, or a particular database schema (note the possibility of multiple definitions in both process and data descriptions). The second type of contents information is a collection of *references* to the uses of the component , i.e. its appearance in a particular DFD or ER diagram. Both types are stored in CDB and are heavily indexed and cross-referenced.

Any of the DFD Edit component manipulations, whether its creation, deletion, renaming or simply re-positioning, is checked for the change in component's information contents and if positively detected, the modification is readily reflected in the CDB database.

Component Types

The CDB contains the logical information of the entire collection of components created as part of the system development. This collection of components consists of:

- *folders* which contain information about different types of projects;
- *documents* which are specific components of the collection of software design documents (e.g., DFDs, E-R diagrams, C code or a Data Dictionary);
- *pages* which are the component of the documents (a collection of pages makes up a document);
- *components* which are the constituent parts of the document being created (e.g., data stores of a DFD, entities of an E-R diagram, field and store sections of a data dictionary, or code modules of a source code).

Each component in the CDB has an associated CId that uniquely determines the component within the database (i.e., the CId acts as a key to the CDB files).

4 Presentation Database (PDB)

Each DFD document, together with its components, has a number of properties which depend heavily on the appearance of the diagram components as defined by the data flow methodology (which varies between software engineering authors), the facilities offered by a particular graphics package, the capability of ten devices, etc.

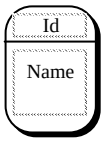
Because of such tremendous dependencies and great variability, the component presentation information is kept separately from their semantic characteristics. The Presentation Database (PDB) is used solely for this purpose.

In this section we give an outline of a PDB structures as dependent on the Hypercard graphic and control system.

Also, due to the similarity in the presentation nature of DFD processes, entities, and stores, in this section we will refer to them simply as *static objects*. Unlike the flows, the static objects' shape and size is predetermined. Data flows and their terminators are considered separately because their shape can not be preset and must be drawn by the designer to fit into the diagram, or generated by the editor.

Processes, Stores, Entities

All of the static objects are constructed using Hypercard buttons and fields of different shapes. Clicking on one of the static object's components causes the execution of its script which will identify it not as a separate card button but as a part of the component itself. Here is a brief description of static objects' layered presentation :



a DFD process is constructed of a base being a round rectangle button, the id and name fields, and a black rectangular button of only 1 pixel height separating the id and name fields;

an external entity is a button styled as a shaded rectangle, with two fields, id and name, placed on top of it;

a data store is implemented as a shaded rectangle button with a 1 pixel wide button separating the id and name fields, in addition an opaque button is placed at the right end of the store to create its "open" look.

Once drawn, the static objects could be easily re-positioned by first hiding all of the auxiliary sub-components (like id and name fields, separation and occluding buttons), dragging their base button to the desired position, and subsequently readjusting the coordinates of all the remaining components and re-displaying them again. The id and name editing can also be easily accomplished by unlocking the fields for this particular purpose then then re-locking them again after the operation is successfully completed.[4]

Flows Terminators

The data flows being *dynamic* DFD components, consist of many editable sub-components, i.e. their line segments, angles, arrowheads, and the name. They are implemented in Hypercard with a set of buttons representing the flow segments, their connectors, and a single field placed across one of the segments and containing the flow id and its name.

[4] It should be noted, that an initial implementation of DFD Edit objects utilised the both bitmap drawing capability of Hypercard mixed together with the use of buttons and fields. This approach resulted in certain inefficiencies, e.g. inability to undo bitmap actions, difficulties in flow editing, etc., thus it was finally replaced by the all button and field solution only.

The connectors represent the arrows of different direction, angled curves, terminators, and the source attachments (vertical or horizontal lines). The choice of angled connectors depends on the segment arrangement. The direction of flow arrows and terminators is determined from the direction of the flow lines.

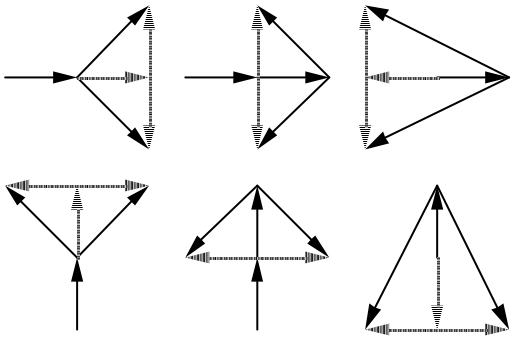


Figure 3 - The construction of flow segments
(black - mouse movements, shaded - actual segments)

The flows are created by first electing the source (static) component and then performing a series of mouse clicks corresponding to the position of the segment connectors. All segments are located immediately under the connector buttons and between the clicks they change their length and direction to follow the grid-constrained mouse movements. The sequence of buttons is terminated upon the click on the target components button or a double click on the background, thus creating a flow terminator.

To restrict the segment directions to horizontal and vertical only all of the diagonal movements of the mouse are translating into a multi-combination of straight segments (ref. Fig. 3).

A flow may be edited by adding an extra segment, which may be accomplished by grabbing one of its segments and then breaking it by dragging. Alternatively we may re-shape it by holding the angle connector and moving it around the work area, all of the segments neighbours will be re-adjusted appropriately. Flows can also be re-linked by holding the source or target attachment (i.e. arrows or straight lines), moving them away from the component, and subsequently adding extra links as during the flows original creation. They may also be re-directed, a double clicking on any segment of the flow causes the change of the flow direction. Double clicking the flow name (similarly to the static object), unlocks their id and name fields for further editing. The position of flow textual information may also be altered by grabbing the flow label and moving it along the flow segments (note the field will "stick" to the flow path).

Component Internal Structures

All of the components, whether static or dynamic objects, have an invisible field associated with their graphical representation. The field contains all of its structural information, i.e. the id-s of the component's constituents, the list of neighbour components, the parent components (if the button is a sub-component of the parent), etc. This field is used while creating and editing DFD components so as to eliminate constant access to the Presentation Database (described in the next section), but also to convey the structural and navigational index information of the DFD, allowing components to be associated with other documents such as data dictionaries.

Presentation Database

The Graphic Database is used to store all the presentation information about each component of the DFD. Each record in this database describes a component and its constituents, which are dependant upon the underlying platform on which the component was created - in this case, Hypercard. The use of this database allows all the presentation information (i.e., the implementation information required by Hypercard) to be separated from the logic information used by the system (e.g., the name given to a process by the designer). The issue of portability is addressed by using logic information to run the system, which is independent of the platform on which the system is built.

Fig. 4 describes the objects used in the implementation of a flow. It shows that the flow named *invoice* with id number 6 is made up of five Hypercard buttons and two Hypercard fields which display the flow id and flow name respectively.

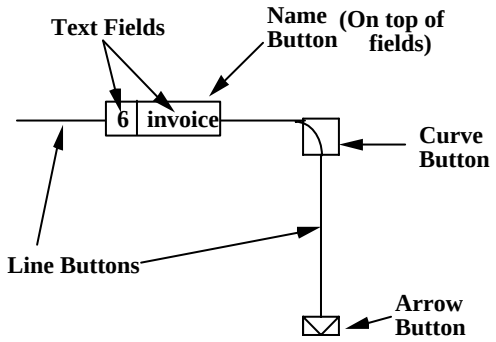


Figure 4 - The Presentation Components of a Flow

Each sub-component of the diagram component is labelled with a number, which we will call a Graphical Id. (GId) number. This number is used to uniquely identify each Hypercard component used in the creation of the DFD. Any event associated with a component (such as a mouseclick on the component) will make use of the component's GId. So, we can view the GId of a component as an interface between the component and the designer. Furthermore the GId can be used as a key index to

a presentation database consisting of all the Hypercard components making up the DFD.

The design of the Presentation Database (PDB) has led to the identification of the types of components to be stored in the database and an analysis of the relationship between these components and the conceptual components stored in the Contents Database. The PDB will store information about the following Hypercard components:

- Stacks which relate to folders (folders may consist of many documents);
- Backgrounds which are associated with documents (e.g., DFDs, DDs,....);
- Cards which are analogous with pages;
- Fields and buttons which map onto sections in pages and various diagram components (e.g., in a DFD, fields and buttons will map onto processes, data stores, etc.);

The four components described above are the entities of the PDB. The relationship between these entities is illustrated in Fig. 5, which is a simplified Entity Relationship diagram for the PDB.

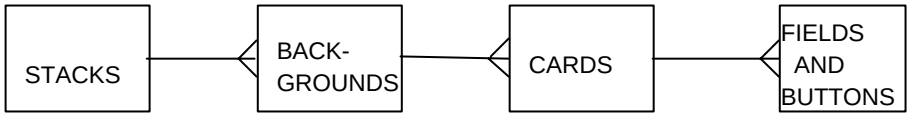


Figure 5 - The Presentation Database Entities

In order to associate the PDB with the CDB described earlier, a cross-referencing table is used, which maps the logic information stored in the CDB to the presentation information stored in the PDB (i.e., this table is the interface between the two databases). The cross-reference table uses the keys of both databases as well as a field specifying the corresponding table in the PDB, to associate records in the 2 databases.

To illustrate the association between the two databases, we refer to the flow described in Fig. 4. The record in the CDB corresponding to this flow is shown in Fig. 6.We noted earlier from Fig. 4 that the flow consists of five Hypercard buttons and two Hypercard fields, so, in the PDB, the flow will be represented by the seven records in the *Fields and Buttons* Table of the PDB, shown in Fig. 6. The association between the two databases storing the flow is represented in the cross-reference table. A mouseclick on the flow will obtain the GId of the Hypercard type

(i.e., field or button) that was clicked on, find the corresponding entry for that GId in the cross reference table, and thus obtain the CId of the flow. Using the CId, the record for the flow in the CDB can be accessed to determine information such as the flows name, etc. Note that until the record in the CDB is found, the component type (i.e., flow, process, data store) of the Hypercard component clicked on can not be determined.

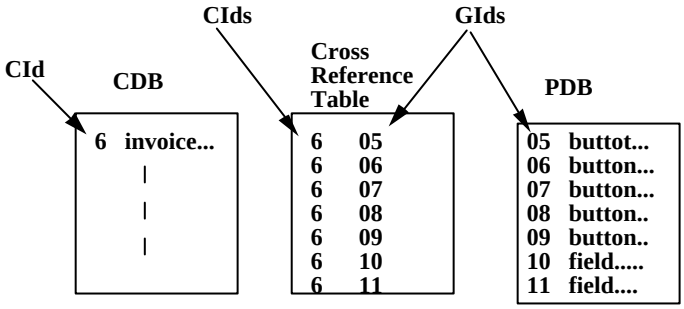


Fig. 6 Association between databases

5 Navigation Database (NDB)

Apart from visually representing the flow of data in a system, DFDs can be viewed as an index to other documents such as data dictionaries and process specifications. Thus each component in the DFD can be used to navigate to various documents containing other forms of information.

Inter Document Linking

Inter-document linking is the core of the HyperCASE project. Its importance is evident when the potential size of the collection of documents describing a system is taken into consideration. A large system would require a huge amount of information processing at the analysis and design phases, resulting in the creation of many pages consisting of parts of DFDs or parts of data dictionaries, etc. The pages must be organised in a way that allows the designer to clearly understand the design of the system, and allows the representation of the documents to move away from the inherently sequential representation of information that is commonly used in books. The use of a hypertext system allows information to be represented and browsed in a logical structure, rather than placing constraints on the way the information is viewed by using a physical structure (e.g. the difficulty in viewing related information that appears in various forms within different documents).

In order to use the hypertext concept in structuring our information, we must first determine how the inter document links (page links) will be established and then associate them with various behavioural properties. Once these properties are

obtained, the link information (i.e., the link along with its properties) will be stored in a database.

Establishing a Conceptual Link

Like the representation of the information content of diagrams, the inter document links make use of the conceptual rather than presentation information of the system. This allows the representation of the navigational links between components to be separated from the underlying platform on which the system is built, thus making the navigation aspect of the system portable over many platforms.

As stated earlier, components can be linked to a number of different documents, each of which may contain the definition of the component (e.g. a process can be defined by a refinement DFD, a Nassi-Schneidermann diagram and a code module). Furthermore, there is no way of knowing how many definitions for a component will, in future, be created and how many component references will appear in the collection of documents making up the project. The linking mechanism must incorporate a method for dealing with these issues which also avoids repetitive information being stored in the Navigation Database (NDB).

The linking mechanism makes use of a transient object (which in implementation view points can be a pop-up menu) to represent all references of a component . Each different type of component is associated with a unique transient object Each time a component is created in a document, the CDB is checked to determine whether a definition of the component already exists. If not, a new transient object is also created and the component is linked to this transient object. Once the component is defined in another document, the transient object is linked to the definition. Thus, all references of the same component , independent of how many documents the component appears in, are represented with one link ,i.e. to the transient object, which in turn is linked to the definition of the component . The creation of any further definitions for the component are simply dealt with by linking the transient object to the definition created.[5]

Presenting the transient object using a pop-up menu allows all the definitions of the component to be visually represented whereby the user can select a desired

---

[5] The method used to establish links has important contributions to integrity checking. If, for example, a process in a DFD has a refinement DFD diagram defining it, other refinement DFD diagrams should not be allowed for that process. The transient object can detect such repetitions.

definition of the component and navigate to it. Each time a new definition for a component is created, it is added to the list of menu items in the pop up menu. When a component is first created and it is not defined anywhere in the system, the pop-up menu is empty, denoting that the component has not yet been defined. Alternatively, the component's definition may be created first whereby the pop-up menu associated with the first occurrence of the component in a document will contain an entry that links the component to a definition.

Establishing a Link at the Presentation Level

All Hypercard components have a script associated with them in which HyperTalk commands can be issued. These commands specify the behaviour of the system when an event, such as a mouseclick, occurs on a component . The script associated with a particular event is known as a handler. A component in Hypercard may have many handlers associated with it, all of which make up the component's script.

In Hypercard, links between components are established using a goto construct in the component's script. A common handler for referencing components is the mouseup handler which performs actions when a mouseup event occurs on the component . A statement in a component's mouseup handler of the form:

```
go to card "data dictionary 1"
```

links the component to a card named *data dictionary 1*. Upon execution of the above HyperTalk statement, the system navigates to the card *data dictionary 1*. The statement is executed when the component receives a mouseup event (i.e., by clicking the mouse on the component).

Hypercard's component linking mechanism is cumbersome in the sense that every link must be specified in the component's script. So, an application with 1,000 linked components would require the specification of 1,000 scripts, each containing a goto statement. Although Apple probably designed this link strategy to allow for modular development of applications, this sort of design separates the links which are related parts of an application and should be represented in a common structure. In order to overcome this limitation, a database has been used to store the links when a component is created and to perform the inter document navigation.

In order to create a link, it is first necessary to determine the type of information required to establish the link. There are basically 2 elements that must be specified in order to create a link:

- the source component ;
- the destination page.

The source component is basically a Hypercard button that is identified by it's name. Similarly, the destination page is a card in Hypercard that is also identified by its name. The link is an association between the source component and the destination page such that when an event such as a mouseclick occurs on the source component, the system navigates to the destination page.

Linking Activities

Depending on the component type that it is linked to, a link may have various activities associated with it. These activities are expressed at the conceptual level and interpreted by the presentation level.

All components making up the documents describing a system can be divided into 2 broad classes; components that have a definition and components that do not (e.g. in DFDedit, processes, data stores, external entities and flows represent those components which have a definition, and terminators, those components that do not). Usually, the definition of a component may also be associated with a definition, and so on. When browsing a document using defined components , it is useful to be able to return to a component once the definition of the component is viewed. This involves keeping track of the browsing path taken by a user and allowing them to traverse the path in reverse. A stack can be used to achieve this, where pages can be pushed on the stack as they are visited and then popped off the stack to return to the pages. This track keeping is not useful for components without a definition and so these components can simply navigate to the page they are linked to without operating on the stack. Note that these components are not associated with a transient object either.

All components can have certain visual effects related to them. For example a terminator pointing to the right can use the Hypercard visual effect scroll left upon navigating to its destination or a process can use the iris open effect when navigating to "expose" it's definition. The visual effect that appears when the component navigates to its destination counterpart is determined depending on the type of component and its properties (which are reflected in the CDB). For example, a terminator's direction can be ascertained by refering to its entry in the CDB and then the appropriate direction for the window scrolling visual effect can be automatically decided.

Pages making up a document usually consist of many parts of information (e.g., a data dictionary has many data definitions on one page). In order to clarify the way

information is viewed throughout the collection of system documents, it is useful to highlight parts of information on a page when navigating to that page. This can be achieved by placing Hypercard buttons with a transparent property over each "chunk" of information stored in the document. Highlighting this button using Hypercard, allows the chunk of information held under the button to stand out from all the other information on a particular page.

All the actions related to components are described at the conceptual level and do not rely on the implementation aspects of the system. This is reflected in the database design in which high level action identifiers are used to represent certain low level activities.

Navigation Database

The Navigation Database (NDB) resides at the logical level of the system (see Fig. 7). It is used to store the links between all the relevant components of the system, communicating with the presentation database (through the cross referencing table) in order to perform the navigation and activities associated with the link, at the presentation level.

A record in the NDB consists of 3 fields specifying the source and destination of the link, and the activity associated with it. Both the source and destinations of the link are represented by their respective Content Id numbers and the activity by a special keyword which is translated by the system to denote a particular presentation action.

In order to access the database, the CId of the component at the source of the link must be known.[6] This can then be used to find the appropriate record in the NDB which specifies the CId of the component representing the destination of the link. We note that the destination field in the database can be a transient object in which case a pop-up menu appears on the screen without any navigation taking place.

---

[6] This can be determined using the GId number -obtained when the component is clicked with the mouse - and the cross reference table, to look up the appropriate CId for the component.

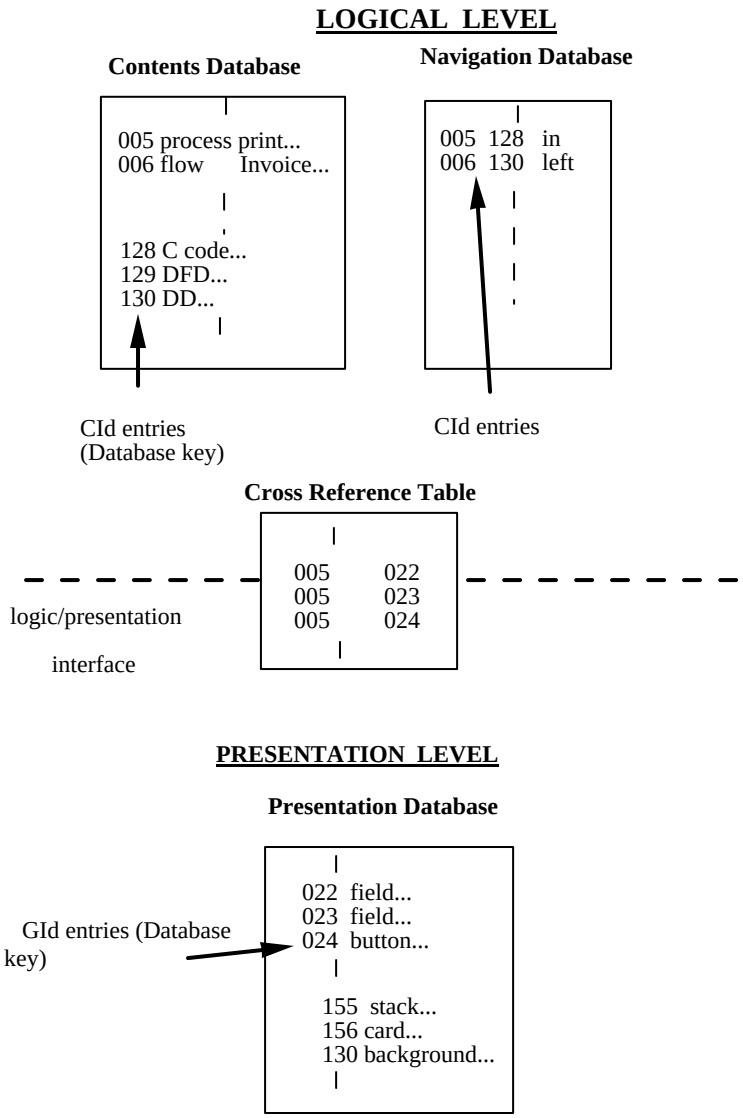


Fig 7 Database separation in the conceptual/presentation levels

A selection of one of the entries presented in the pop-up menu determines the destination component of the link and another search through the NDB locates the record representing this link. This serves only to determine the activity associated with the link since the destination of the link was obtained when the user selected an item from the pop-up menu.

6 Conclusions

Summary

This paper describes a prototypic diagram editor, DFD Edit, being an inherent part of the HyperCASE user interface, used in the construction of software diagrams with a particular emphasis on Data Flow Diagrams. The editor has incorporated the hypertext concept by allowing all diagram components to be represented as

hypertext buttons, facilitating their linkage with various other documents containing different forms of information. A commercially available hypertext system, Hypercard developed by Apple Computer, Inc., has been used as the platform on which the prototypic editor has been implemented.

Special emphasis was placed on the separation of the conceptual and presentation information of the editor, thus allowing it to run independently of its implementation aspects. A database for the editor has been designed that incorporates the separation, and overcomes some of the problems related to efficient utilisation of space.

Finally, various factors related to the development of diagram editors have been considered. Aesthetics have been determined to assist the development of neatly laid out diagrams, which have been incorporated using a grid technique. Furthermore the representation of diagram components as buttons has been accomplished which allows them to be easily manipulated and linked to various other documents.