Browsing the Chip Design Database

David Gedye and Randy Katz

Computer Science Division
571 Evans Hall
University of California at Berkeley
Berkeley, CA. 94720

ABSTRACT

A *design browser* is a tool for exploring the interconnected web of design objects managed by a CAD database. The browser described in this paper is the first such tool to present this information graphically — directed graphs are drawn to show the relationships that exist between objects in the database. Since graphs can become very large, techniques referred to as *rectangular* and *hourglass pruning* have been developed to reduce the information displayed to a manageable level. This browser allows designers to visualize the rich, multidimensional structure of their designs.

November 8, 1987

Browsing the Chip Design Database

1. INTRODUCTION

A *design browser* is a tool that allows engineers to explore the high-level structure of their chip designs. When these designs are managed by a powerful CAD database, a rich, interconnected web of design objects will exist. It is on this web that a design browser operates.

Browsers are important in such an environment because they reveal unobserved, or forgotten structure, enhance design re-usability by making previously designed components easy to find, and assist in communication and shared understanding between different members of a design team.

The composition hierarchy of a design is an example of a high-level structure that might be browsed. Depending on the sophistication of the CAD environment other relationships (such those managing version control) may exist between design objects and can also be the target of a browser.

There is an obvious and tight connection between a design browser and its underlying data manager. The objects and relationships that a browser presents must be drawn from this database, and in the case of the research reported in this paper, the underlying data manager is the **Version Server** [Kat87a,Kat87b] currently being developed at UC Berkeley.

The significance of our browser is that it is the first tool to present the graph structures in a design *visually*, as two-dimensional pictures. Most CAD data managers provide a mechanism to traverse a graph node-by-node, and edge-by-edge, but so far no other tool described in the literature lets an engineer view an entire graph at a time. However, similar pictures have been produced in other fields. Some hypertext systems display the links between a set of documents graphically, and many object-oriented programming environments allow a user to display an object hierarchy as a directed graph.

As CAD data models become richer and more complex, questions like *Where is component A used, and what other components may be affected if I make a change to it?* become increasingly difficult for the engineer to answer without a powerful interface to the data manager. Our browser cannot solve this problem by itself. It does not model the geometric, electrical, or behavioral properties of an IC design, but its contribution is that it addresses the structural aspects of the query in a new and powerful way. For the first time, designers can 'see' the high-level relationships between the objects in their design database.

The browser described in this paper is closely coupled to a VLSI CAD environment, but the idea of graphically representing and manipulating structure is of value in almost any design field. Whenever there are a large number of objects to be maintained, and it is important that the human designer understands the relationships between these objects, a graphical browser is the obvious tool.

This research is supported by the National Science Foundation under grants MIP 8706002 and MIP 8352227, with industrial matching support from the Microelectronics and Computer Technology Corporation.

The paper is organized as follows. Section 2 explains the Version Server data model; Section 3 presents the main visual features of the browser; Section 4 explains how engineers can use the browser to *navigate* through the inter-connected web of objects in a design database; Section 5 describes *graph pruning*, our method for dealing with structures too large to be comfortably displayed as a whole; Section 6 addresses the performance (speed) of the browser; Section 7 summarizes the main implementation issues; and Sections 8 and 9 discuss related and future work.

2. THE DATA

The Version Server CAD Data Manager[Kat87a] maintains an 'arms-length' view of the chip design process. It does not concern itself with the representation specific data (layout geometry, netlists, functional descriptions, simulation decks, test vectors, and so on), but focuses on a small number of structural relationships between abstract *design objects*.

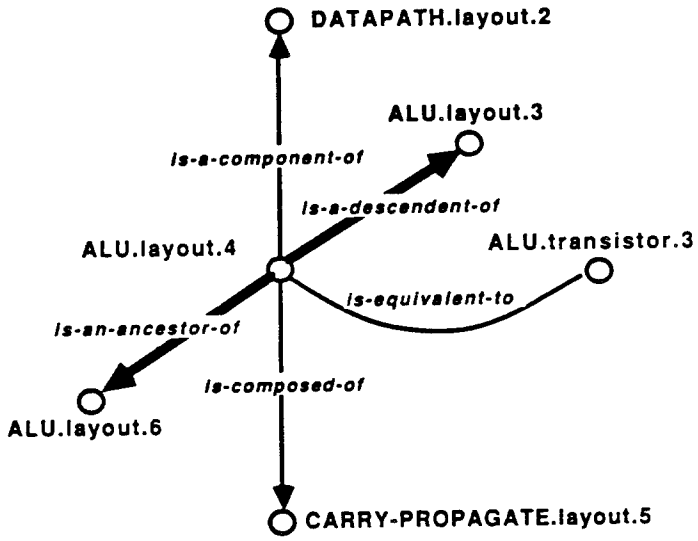

Figure 1. The relationships supported by the Version Server. 'ALU.layout.4' is a component of (among other things) 'DATAPATH.layout2', and is composed of (among other things) 'CARRY-PROPAGATE.layout.5'. Similarly, it is a descendant of 'ALU.layout.3' and is an ancestor of (among other things) 'ALU.layout.5'. Finally, it is known that 'ALU.layout.4' is equivalent to (among other things) 'ALU.transistor.3'

Figure 1 illustrates the 5 types of relationships that a design object can participate in. Two of these relationships (*is-a-component-of* and *is-composed-of*) model the hierarchy of a chip design as a directed acyclic graph (DAG); another two (*is-an-ancestor-of* and *is a descendant of*) provide the links needed to maintain each design object in a tree of versions of the same component, and the final one (*is-equivalent-to*) partitions the space of design objects into equivalence classes, as determined by the designer or perhaps a tool.

With one exception, these relationships are not 1-1. A design object can be composed of many other objects, can be a component of many objects, can be an ancestor of many objects, and be equivalent to many objects. However, the important restriction in the Version Server's data model that an object can only have one immediate ancestor restricts version graphs to trees.

It should also be noted that the three sets of relationships (composition, version and equivalence) are orthogonal; version information is independent of composition information, which is also

independent of equivalences. This leads to a very rich web of interconnection between the design objects. In effect, each object participates in three unrelated data structures — a composition DAG, a version tree, and an equivalence set.

Design objects are named in a standard three component form in the Version Server. 'ALU.layout.4', for example, should be read as version 4 of the layout representation of ALU. In many of the figures which follow the browser's abbreviated naming scheme has been chosen to compress redundant information. All design objects in a version tree, for example share the same initial two components, and these may be dropped from the display with no loss of information. By similar (but not quite so thorough) reasoning the names of design objects in composition hierarchies are abbreviated to just their first component, and those participating in equivalence relationships to only their second component.

The high-level structures maintained by the Version Server allow designers to answer queries like:

> *Where did I use the multiplier I designed last week?* (A composition query.)
>
> *Show me the basic multiplier I modified when I designed the new one last week?* (A version query.)
>
> *Do we have a schematic corresponding to the new multiplier, and if so where?* (An equivalence query.)
>
> *Does the behavioral description of the multiplier that I wrote last month still correspond to my current implementation?* (An equivalence query, relying on the change propagation features of the Version Server [Kat87b].)
>
> *Does our netlist model for the multiplier have the same hierarchical structure as the layout, and if not where do they differ?* (A query involving both equivalence and composition.)
>
> *Do all the versions of the layout of the multiplier have corresponding timing information?* (A query involving both versions and equivalences.)

A browser that attempts to answer queries like these graphically is faced with two immediate problems. The first is that the number of relationships that any given object participates in can be large. For example, a low-level component in a chip design, say the layout of a multiplexor, may participate in tens of *is-a-component-of* relations, and several each of *is-composed-of*, *is-an-ancestor-of* and *is-equivalent-to*. The challenge is to display this information about the particular design object, without the result becoming a tangled mess.

The second problem confronting a browser is that the raw number of objects can be very large. A composition hierarchy may contain hundreds or even thousands of objects, and so any attempt at showing the entire structure, even for a single relationship type, is quite infeasible. Even to display all the objects which bear a particular relationship to a single given object (say, the tens or hundreds of cells which make use of a standard I/O pad) can prove difficult.

The first of these problems is addressed in the basic design of the browser, which is described in the next Section, and our solution to the second problem, *graph pruning*, is explained in Section 5.

3. THE VISUAL DESIGN OF THE BROWSER

The browser presents its data as directed graphs drawn on the screen of an Sun workstation running the Sunview window system[Sun86]. Component hierarchies are displayed in exactly the way engineers usually think of them; as DAGs where the nodes are components and the edges signify containment. Version trees are drawn as trees, and an equivalence set (which has no commonly agreed on graphical representation) is displayed as a group of named objects in a window.

Multiple windows are used to simplify the presentation, as shown in Figure 2. These windows are *typed* — apart from the upper control panel each window is restricted to showing either composition relationships, version relationships, or equivalence relationships, but never a mixture.

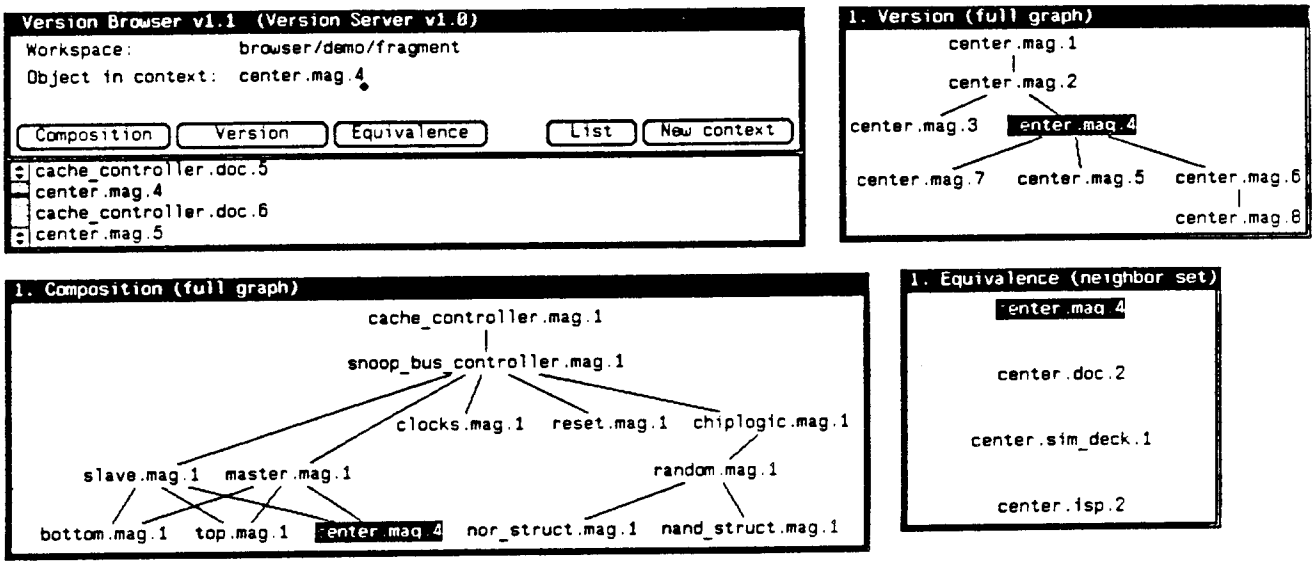

Figure 2. A screen-image of a browser context-group. The control panel in the top-left window allows the designer to specify the particular workspace to be browsed, and to type in the name of a design object to be investigated. The lozenge-shaped buttons at the bottom of the control panel are for creating new windows. The 3 left-most buttons create the windows that are visible at the bottom of the figure, while the 'New Context' button creates an entirely new browsing environment, usually displaying a different object, and potentially accessing a different workspace.

Edges in the directed graphs are displayed as simple line segments. The directionality of the edges is implicit by the relative positions in the window of the objects being connected — all edges point down-hill. (It is possible to simplify the display in this way precisely because all the graphs are acyclic.)

Each of the lower windows shows information about the same highlighted object, the *object in context*. The composition window shows it in relation to other objects in its composition DAG, the version window shows how it is related to other versions of the same real-world object, and the equivalence window shows the other objects that have been deemed equivalent to it.

The collection of windows displayed in Figure 1 is referred to as a *context-group*, since it is focused on exactly one design object at any particular time. As the engineer navigates through the structure (in a manner described in the next section), all the windows in a context-group

update themselves together. In general they will be displaying different information, but it will always be information about the same design object.

Sometimes it is desirable to be browsing in more than one context-group at a time. This can be very helpful if the engineer is trying to compare two structures, for example two different composition hierarchies, as is the case in Figure 3. New context-groups are created by selecting the appropriate command button in the control panel of any existing context-group.

The windows in Figure 3 are grouped together according to context-group, but this is completely at the discretion of the user. The position, size and shape of every window on the screen can be modified by the user at any time. Often it makes sense to split up context-groups by moving windows from different groups close together so that they may be compared better.

A simple 'clean up' function is provided in a pop-up menu in the context window. Windows belonging to that context-group are marshalled together, and placed in a strip below the context window.

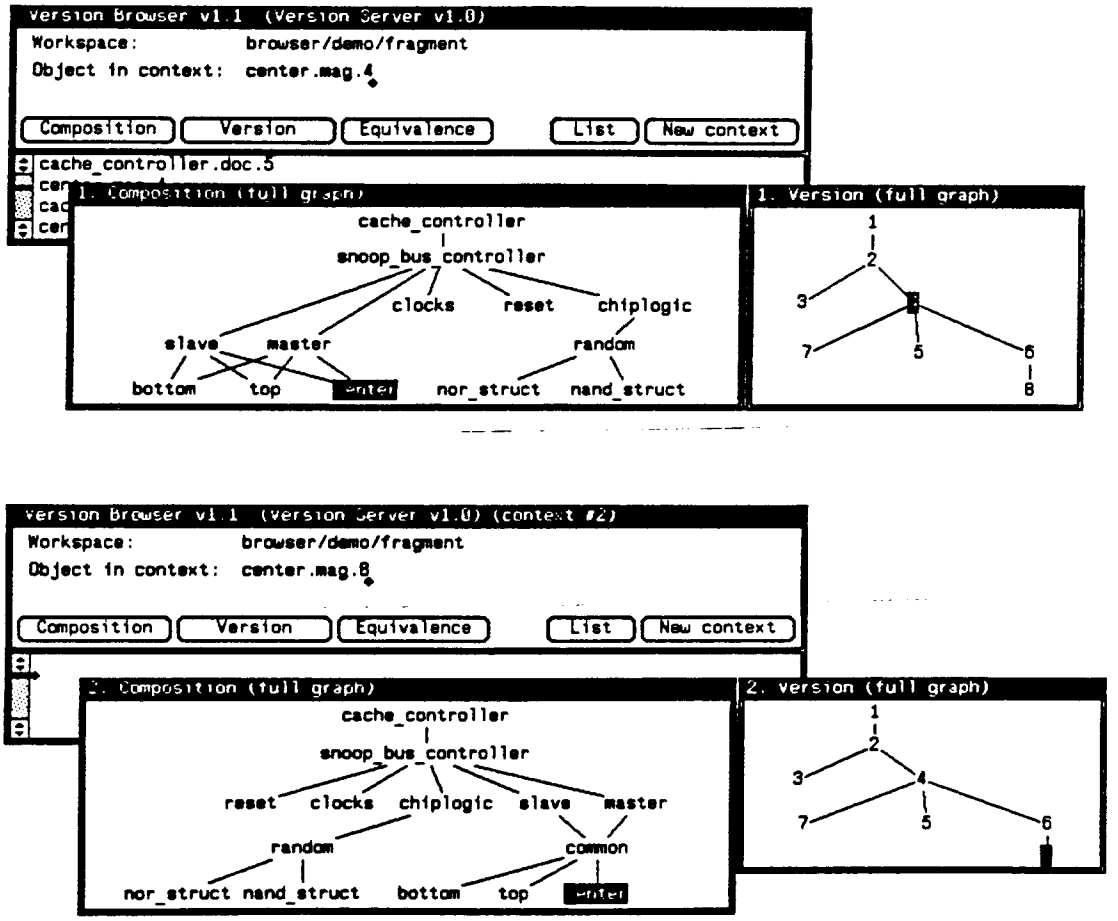

Figure 3. A browser with multiple context-groups. In this example two composition hierarchies corresponding to the first and second versions of a design for a cache controller are being compared. Context-group 1 is focused on 'center.mag.4' and context-group 2 is showing 'center.mag.8'. Note that although the composition hierarchies which include in these two versions of the 'center' object, they appear as different nodes in the same version tree — they are both versions of the same real-world object.

4. NAVIGATION IN THE BROWSER

Since the main goal of our browser is to let engineers 'explore' their design data easily, powerful mechanisms for navigating through the multi-dimensional graph are essential. The task of specifying the next object in context must be simple and fast.

One general way to specify a design object is to provide its name. A type-in field in the control panel of each context-group exists for this purpose. When the user types in the name of a design object in the current workspace all windows associated with this context-group immediately update themselves. Depending on their type, they display composition DAGs, version trees, or equivalence sets centered on the new object in context.

If the exact name of a particular design object is not known it can be found in the scrolling, text buffer at the bottom of the control panel. When the 'List' button is pressed, the names of all the objects in the workspace are written into this buffer, and all the usual manipulation features of a text editor (such as searching for a string) are available.

The mechanism described above is general, because any object in the workspace can be specified by its name, but it fails to take advantage of locality. When an engineer is browsing the structure of a chip design there is a high probability that the *next* interesting design object will bear some relation to the *last* one. For example, after studying the version tree associated with the schematics of a component, a possible next step might be to turn to the version tree of some equivalent representation of the component, say its behavioral description, and compare the two. In this case, the relationship that the two successive objects in context would be *is-equivalent-to*.

To exploit this observation a second navigation mechanism was added to the browser: any design object in any window can be selected with the mouse, and immediately becomes the new object in context. Windows that already contain the new object simply change their highlighting, and all others update themselves completely. The mechanism, which we call *local stepping*, is simple, fast and obvious. If you are curious about any object you can see, simply press the mouse button over that object, and it becomes the object in context.

To illustrate this, consider Figure 3 again. The situation in context-group 1 is a single local step away from that in context-group 2. Selecting 'center.mag.8' (represented by '8') in the version window associated with context-group 1 would bring the same object into context in both sets of windows. The graph in the composition window of context-group 1 would be discarded, and a graph identical to the context-group 2 composition graph would be drawn in its place. The same process will occur in the equivalence windows.

5. PRUNING THE GRAPHS

Unlike Figures 2 and 3, real composition hierarchies often contain hundreds of design objects. To be of use to a designer involved in a medium or large chip design project, a browser must be able to cope with hierarchies of this size. (In theory, version trees and equivalence sets could also become very large, but in practise this does not happen. Typically there are a handful of versions of any particular component, and a similar number of equivalent representations.) Figure 4 shows the entire composition DAG of a large chip design currently being undertaken at UC Berkeley. It is clear that, as it stands, this is not a particularly useful presentation of the structure.

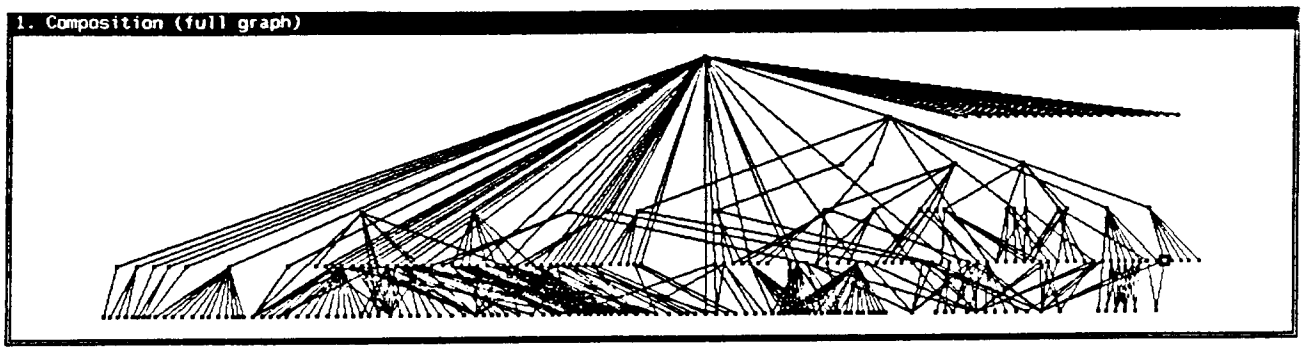

Figure 4. The composition hierarchy of the SPUR [Hil86] Cache Controller chip. This graph contains 267 nodes and 431 edges.

5.1. Standard Approaches

The problem of having too much graphical information to fit comfortably on the screen is not unique to browsers. Layout and schematic editors, for example, have always been faced with this problem, and it is instructive to examine the standard solutions before explaining why a new approach is needed for a graph-based browser.

There are two mechanisms commonly found in graphical editors to reduce the amount of data on the screen to a manageable level.

- *Zooming.* The user is given control of a scale parameter, and depending on the value chosen, either a small amount of the design is seen in detail, or a large amount is seen in overview. This reduction in detail comes automatically when the resolution of the output device degrades the quality of the graphic. (For example when the entire layout of a VLSI chip is painted onto the screen of standard workstation.) Often however, the program will take advantage of structure in the data to simplify the display of complex objects as they it zooms out. In a hierarchical editor for example, this is often done replacing the contents of some object by its 'black box' representation. This technique is already used in a limited way in the browser. The names of design objects can be abbreviated from their standard three component form, or dropped altogether and replaced by dots, as has been done in Figure 4.
- *Panning.* When the editor is zoomed-in, and only a portion of the the edited design being displayed, the user must be given some control of *which* portion should be displayed. In a layout editor this choice is usually drawn from a continuum of possibilities. Any (x,y) position in the design can be shown at the center of the screen; like a camera, the editor *pans* to the chosen position and displays some region surrounding that point. Schematic editors usually also offer discrete panning. Large schematics are commonly broken up into managable 'pages' when they are being designed, and it is common to display only a single page at a time.

Unfortunately, neither of these mechanisms can be used in any significant way in the browser.

Since a composition DAG has no hierarchy of its own (it *represents* a hierarchy, but in itself it is a flat data structure) it is not clear how parts of the graph could be 'collapsed' together in any meaningful way to affect a zooming operation. The names can be abbreviated, but this does not reduce the raw number of objects to be displayed.

Conceivably, a browser might zoom in on a graph like Figure 4, and pan the window to bring the object in context to its center. The result of such a well-intentioned operation would generally be quite unacceptable. Closely related design objects may, or may not, be included in the rectangular window; totally unrelated (and at present, uninteresting) objects may consume much of the space; and stray edges connecting nodes completely outside the window may cut distractingly across the display. This is not surprising, for the layout algorithm attempts to optimally display the entire graph, not just the particular subgraph of interest at the moment.

If the goal is to show only *part* of a composition hierarchy, then clearly that part should be selected first, and *only* that part layed out as a graph in the window. This will avoid all the problems of extraneous nodes and edges interfering with the display. The subgraph chosen should clearly include the object in context, but just which other nodes and edges should also be displayed is not so obvious. We refer to the process of choosing such a subgraph as *pruning* the graph.

The goal of pruning is to select a small enough set of objects 'closely related' to the object in context, so that they can be clearly displayed in the available space. Ideally, we would like to choose those objects which are most relevant to the engineer's understanding of the object in context, but this is not a simple matter.

The two algorithms described below, *rectangular pruning*, and *hourglass pruning*, differ only in their determination of what 'closely related' means. Neither captures the concept completely, but we feel that they are an obvious place to start and the selections they make are useful. We are still exploring other pruning strategies.

5.2. Rectangular pruning

The basic mechanism involved in both pruning strategies is bounded, breadth-first search, starting at the new object in context. In a composition hierarchy this search proceeds both forward along *is-composed-of* edges, and backwards along *is-a-component-of* edges.

Every node encountered in the traversal of a directed graph can be assigned a *level number*. The starting point is assigned level 0, its immediate sucessors level 1, their immediate sucessors level 2, and so on. Immediate predecessors of the starting node (those reached by a backwards edge traversal) are assigned to level -1, and so on.

Rectangular pruning takes the candidate nodes in the order in that they are provided by the breadth-first search, and selects a rectangle of nodes from this levelized graph. It is controlled by three parameters: **up**, **down**, and **width**. **Up** is an upper bound on the number of levels to consider above (negative level numbers) the object in context; **down** is the maximum number of levels to consider below (positive level numbers) the object in context, and **width** is the maximum number of objects that should be displayed in any one level. Figure 5 illustrates rectangular pruning.

The rationale behind rectangular pruning is that as many as possible of the closest nodes should be included, with preference going to the direct ancestors, and direct descendants of the object in context. In this algorithm the 'closest' nodes are exactly those that are the fewest number of edge

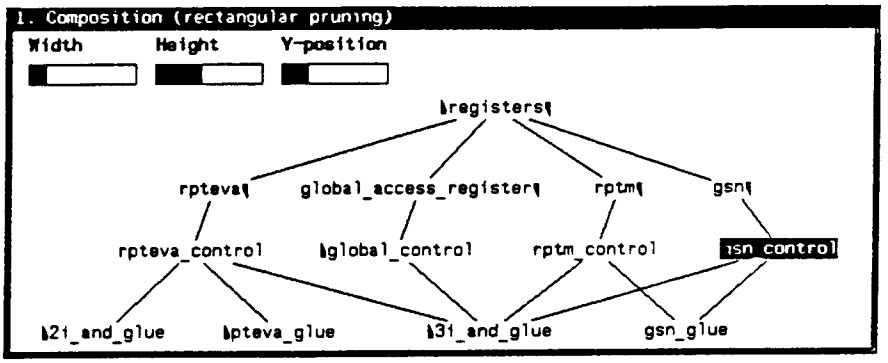

Figure 5. An example of rectangular pruning. The object in context is gsn_control and the up down and width parameters are 2, 1, and 4 respectively. The oblique symbols before and after some of the object names are called *pruning marks*, and inform the user that objects directly related to the object on which they appear have been pruned away. Upwards pointing marks imply that at least one ancestor is missing, and downwards pointing marks imply missing descendants. The design object 'registers' has pruned ancestors because of the value of the up parameter, and pruned descendants due to the value of the width paramter.

The analog bar controls in the top-left corner of the window allow the user to control the values of up, down and width. The first two parameters are not manipulated directly. In their place composite parameters height, and Y-position are offered. Height equals up + down + 1, and corresponds to the overall height of the rectangle. Y-position equals "up + 1" and signifies whether the object in context is to be placed near the top of the window, in the middle, or near the bottom. In effect, the Y-position controller scrolls the subgraph up and down in the window.

traversals away from the object in context. The preferential treatment of direct relatives follows automatically from the order properties of breadth-first search. The first nodes encountered at any level will always be the direct descendants or ancestors of the starting node.

5.3. Hourglass Pruning

Hourglass pruning is similar to rectangular pruning, but it specifically excludes any object from consideration that is not a direct ancestor or descendent of the object in context.

The only object at level 0 will be the object in context, but adjacent levels can fill up to the maximum allowed width with parents and children of the object in context. The pruned graphs may thus assume an hourglass shape, with the waist of the hourglass at level 0.

Often in chip design it is important to know all the components, and sub-components of a design object, or perhaps all the higher level objects which include the given object as a component. Hourglass pruning exactly captures this set, and does not attempt to display any object not meeting one of these criteria. Figure 6 contrasts rectangular and hourglass pruning.

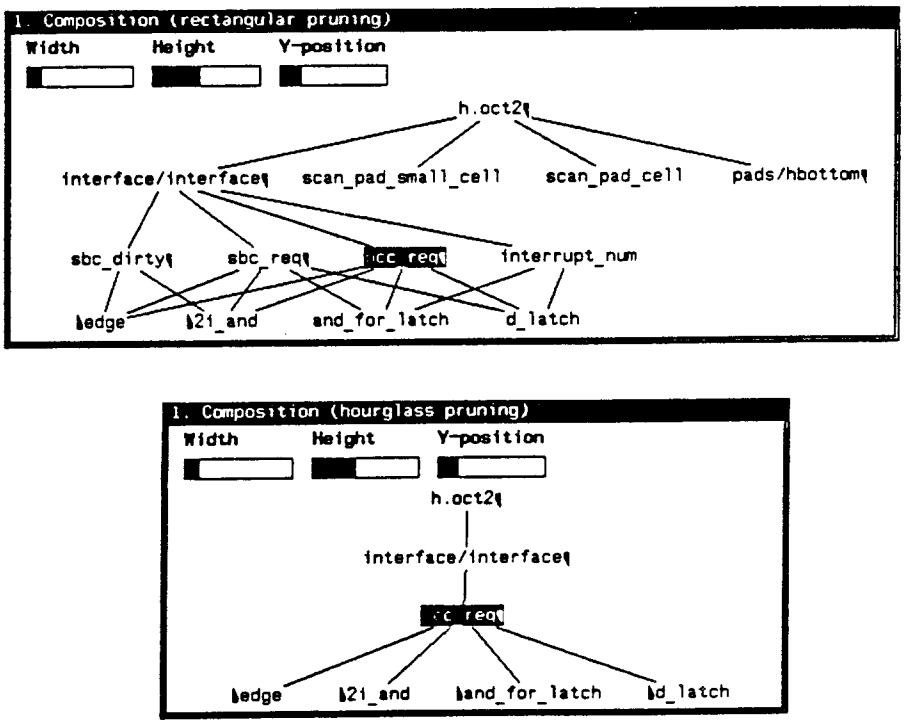

Figure 6. A comparison of rectangular and hourglass pruning. In both cases the parameters 2, 1, 4 were used for up, down and width, but the hourglass algorithm includes many fewer objects. 'Siblings' of the object in context, such as 'sbc_dirty' and 'sbc_req' have been removed, as have 'uncles' such as 'scan_pad_small_cell' . Only direct ancestors and descendants remain. This figure also illustrates the fact that sometimes even the object in context may have to be pruned. In this example 'pcc_req' has more than 4 descendants, and so some of them cannot be displayed without increasing the width parameter.

5.4. User Interface to Pruning

Users can control the results of pruning in three distinct ways. Firstly, they can use a pop-up menu to choose the style of pruning (currently, rectangular or hourglass). Secondly, they use analog control panels to determine upper bounds to the number of objects returned by the pruner, and the overall shape of the pruned graph. Finally, they can use the standard window system operations to change the size and shape of a window, so that the pruned graph fits comfortably in the window. Since window sizes can be adjusted at any time, the standard procedure is to see what the pruned graph looks like, and adjust the size and shape of the window to suit it.

6. PERFORMANCE OF THE BROWSER

The critical performance measurement for a highly interactive tool such as the browser is response time. To support free-ranging exploration through the web of design objects it is essential that the basic operation of changing context (initiated either by naming the new object in context, or pointing to it) should complete within a small number of seconds. If this were not the case, users might be hesitant to explore the 'what-if' scenarios that the browser was intended to address.

The design goal for performance was that, under normal circumstances, context changes should take no more than two seconds from the moment the mouse button is released to the moment the last window in the context-group completes its update. This goal has been met.

'Normal circumstances' are as follows: execution on a Sun 3/140 workstation; three windows per context-group — one each for composition, version and equivalence relationships; and no more than a total of 50 design objects to re-position on the screen as a result of this operation. The two context-groups in Figure 2, for example, each contain 2 layout windows and roughly 20 design objects. It takes less than a second from the time the '8' object is selected in the version window of context-group 1 until the both windows are updated to match those in context-group 2.

Searches through large workspaces are not inherently slower than through small ones. As long as pruning is employed so that the number of objects to be displayed is bounded by a small constant, context changes will occur swiftly. The context change which produced the composition window in Figure 5, for example, required less than a second to compute.

Windows attempting to display very large unpruned graphs, however, do take substantial time to update themselves. The 267 node graph in Figure 4 took 32 seconds to compute on a Sun 3/75 We feel that this is acceptable given that such large displays, in which the object names must be omitted, are much less useful to the designer than the small, fast, pruned subgraphs.

7. IMPLEMENTATION OF THE BROWSER

The browser is a C-language program and runs on the UNIX™ operating system. It makes use of the following three independent software packages, all of which are linked together with the browser, and run as a single process.

- *The Version Server*. To a client program, such as the browser, the Version Server CAD database [Kat87a] is simply a set of C-language functions. The Version Server provides functions to find a given design object by its three-component name, and to locate the objects that a given object is directly related to. The bounded breadth-first search, described in Section 5, consists of a single call to the first function, followed by repeated calls to the second function, during which the browser builds up its own data structure to represent the subgraph to be displayed.
- *Sungrab*. The Sungrab Graph Layout Package [Row87], developed at Berkeley, is another set of C-language routines. It reads (or is passed) a specification for a directed graph, and it attempts to find a two-dimensional layout of the graph which minimizes the number of edge-crossings. The 'attractiveness' of the graphs in Figures 2, 3, 5, and 6 is due completely to the sophistication of Sungrab.

  It should be noted that pruning is not considered part of the graph layout task, and is not handled in Sungrab. Routines in the browser which interface to the Version Server implement the pruning *during* the breadth-first seach, and only the appropriate subgraph is passed to Sungrab.

  The the graph layout package is the only computationally significant component of the browser*. It accounts for at least 80% of the context change time in very large graphs such as Figure 4.

* This may change when more sophisticated pruning strategies are employed.

- *Sunview*. The Sunview Window System [Sun86] is a kernel based window and graphics manager. It provides interface functions to create and manipulate windows, render the layed-out graphs onto the screen, and associate pop-up menus with windows. in a window.

  In particular, the Panel Package within Sunview provides simple object-oriented support for active items. When a *panel item* is created the client (in this case the browser) can associate with it a set routines that Sunview will call whenever this panel item is selected by the user. Every node in the every graph is implemented as a Sunview panel item. This decision moved the point-location problem *(over which object was the mouse button clicked?)* from the domain of the browser, to that of Sunview; a significant saving in programmer effort.

The attraction of Sunview's panel package, together with the general maturity of the toolkit and quality of documentation, were the main reasons why Sunview was chosen as the underlying window system rather a public domain package such as X [Sch86]. The disadvantages of Sunview for this project were its proprietary nature — the browser cannot be easily ported to other vendors' workstations, and its kernel based architecture — compared to a network oriented window system like X, browsing remote databases is very difficult.

The principle of software modularity dictated that the three packages should be as kept as independent as possible. This strategy had disadvantages as well as advantages. While it simplified understanding of the program as a whole, and reduced implementation time correspondingly, it led to a potentially troublesome proliferation of data structures. Each design object displayed in a window, for example, is modeled by four independent data structures: one maintained by the browser itself, one used internally by the Version Server, one used by Sungrab in laying out the graph, and the panel item used by Sunview to represent the object in the window. These data structures each hold different information about the design object, but consistency between them must still be maintained.

8. RELATED WORK

No research reported in the commonly available literature addresses the problem of graphically browsing CAD data. The browser described in this paper belongs to the synthesis of the two fields of CAD database querying, and general graphical browsing. This section discusses both areas.

8.1. Querying a CAD Database

Queries to a CAD database are often more complex to specify and implement, than queries to a general purpose database. This is the because the underlying structure supported by CAD databases is generally richer than the common relational model.

Full support for component hierarchies, for example, implies that a query mechanism involving graph traversals should be provided. The current browser is particularly simplistic in this regard. Almost the *only* query that a user can make amounts to *return the set (or a pruned subset) of the design objects which, by transitivity, bear a particular relationship to the object in context.* Other query processors, such as the one in development at Computer Corporation of America [Ros87] are much more sophisticated in this regard. An example of a query in this system might be *determine the power dissipation of component X.* This query implies a traversal through all the components of X, a computation done at each node, and a cumulative summation maintained throughout.

Pruning can be an important part of these queries. For example, a query such as *return all components larger than 1000 square microns* should traverse the graph examining component sizes, and prune away all subgraphs rooted at a design object smaller than the given size.

Languages to express such complex queries are generally textual, but interface improvements originating in general purpose query systems have been successfully applied. G-WHIZ [Hei85], for example, provides a forms-based, QBE-style interface to a CAD database.

Certain interfaces, such as the one provided by Attache for the Oct [Har86] data manager, stress the activity of local browsing, rather than global queries. In Attache, as with the current browser, there is always an object in context, and all displayed information relates to it in some way. Unlike our browser, however, the *only* information on the screen is that which directly describes the object in context. This can present difficulties in navigation, because unless the next 'interesting' object is directly related to the current one, it will not be present on the screen to select, and cannot be specified simply by clicking a mouse button.

8.2. General Graphical Browsers

Whenever data is modeled as a collection of objects and relationships, there is the potential for browsing it graphically.

This potential has been well exploited in the field of object-oriented programming environments, such as Xerox's Interlisp-D [She83]. In Interlisp-D, which runs on Xerox's bitmapped workstations, the object type hierarchy can be graphically browsed as a directed acyclic graph. Types are represented by nodes in the graph, and the *IS-A* relationships by edges. Object graphs run left to right, rather than top to bottom as in the current browser. General pruning is not supported, but any type object can be treated as the root of the DAG for the purposes of browsing. This limits the displayed graph to the selected type and its sub-types.

Hypertext systems [Con87] are another domain in which graphical browsing is used sucessfully. Both Neptune [Del86] and Intermedia [Gar86] provide browsers for graphs consisting of the hypertext documents and the links that connect them. In hypertext systems the usual semantics associated with clicking the mouse button over a node is to open an editing window for the corresponding document. Currently reported hypertext browsers do not perform dynamic pruning, but Intermedia has a proposed static mechanism that addresses the same problem. Each link is assigned to one or more *webs* when it is created, and the user can choose to examine just the objects linked in to a particular web.

Software development environments have also made use of graphical browsers. In the Pecan Program Development System [Rei85] program parse trees and control flow graphs are browsed graphically. Users can examine these graphs statically, or watch an animated simulation of the executing program — nodes in the flow control graph highlight themselves as the corresponding statement in the program is executed. Large graphs occur often in this system, but pruning is not employed. Window scrolling (equivalent to the panning operation described in Section 5) is used to examine these large graphs closely.

Finally, there are application-independent graph browsers, such as GRAB* [Row87]. GRAB reads in a specification of nodes and edges from a file, lays out the entire graph using a sophisticated algorithm that reduces the number of edge crossings, and displays the graph in a scrollable

* GRAB was developed at UC Berkeley by Larry Rowe and his students, and its graph layout code was borrowed for the browser described in this paper. We acknowledge a significant debt.

window. It is easy to interface other UNIX tools to GRAB, so that the effect of clicking the mouse button over a node in the graph is to start the tool (often a text editor) on some corresponding file.

9. FUTURE WORK

There are a great many ways in which the current browser could be extended and improved. We are currently working on several of the ideas listed below.

- *The Browser as a front end to the CAD environment.* We intend to extend the browser so that it can invoke Version Server operations and specific CAD tools on selected design objects. We see this as a straightforward extension of the current work.
- *Attribute-based browsing.* Designers often do not know the exact name of the component they are interested in browsing, but can easily specify it by a combination of attributes. An interface which could handle commands like *make the fastest multiplier the object in context* would be extremely useful, but appears to require a textual command/query langauge — something completely absent in the current browser. Attribute-based browsing also requires attribute support in the underlying data manager, and as yet the Version Server does not provide this.
- *Domain-based pruning.* The pruning strategies described in Section 5 are relatively primitive, and syntactic in nature. They do not take advantage of the fact that these graphs represent component hierarchies in any significant way. We propose to extend the browser to allow *domains of interest* to be defined. These would be subsets of design objects considered to be a natural unit by the designer. They would be specified by an attribute-based predicate, such as *all objects used in the cache controller which have been modified in the last 4 days,* and could be integrated into the pruning system. Only those objects which satisfy the predicate are considered for display, and of these, only the ones 'closest' to the object in context (as determined by a pruning algorithm) are actually displayed. Domains of interest are similar to the Intermedia *webs*, mentioned in the previous section.

10. SUMMARY

This work has demonstrated that it is both feasible and worthwile to graphically browse the contents of a CAD database. Designers can quickly and easily navigate through the rich web of design objects to answer simple structural queries. When the web becomes too dense to usefully display in a window, dynamic graph pruning algorithms strip away parts of it which do not directly relate the object currently under investigation.

11. REFERENCES

[Con87] J. Conklin *Hypertext: An Introduction and Survey.* IEEE Computer, Vol. 20, No. 9. September 1987.

[Del86] N. Delisle and M. Schwartz *Neptune: A Hypertext System for CAD Applications.* Proc. ACM SIGMOD Intl. Conf. on Management of Data, Washington D.C. May 1986. pp132-143.

[Gar86] N.L. Garrett, K.E. Smith, and N. Meyrowitz *Intermedia: Issues, strategies, and Tactics in the Design of a Hypermedia Document System.* Proc. Conf. on Computer-Supported Cooperative Work, MCC Software Technology Program, Austin, Texas, 1986

[Kat87a] R. Katz, R. Bhateja, E. Chang, D. Gedye and V. Trijanto, *Design Version Management* IEEE Design and Test Magazine, Vol 4, No. 1, February 1987.

[Kat87b] R. Katz, E. Chang. *Managing Change in a Computer-Aided Design Database.* Proc. 13th Intl. Conf. VLDB, Brighton, England, Septmeber, 1987.

[Har86] D. Harrison et. al. *Data Management and Graphics Editing in the Berkeley Design Environment.* Proc. ICCAD, Santa Clara, November 1986. pp 24-27.

[Hei85] S. Heiler and A. Rosenthal. *G-WHIZ, a Visual Interface for the Functional Model with Recursion.* Proc. 11th Intl. Conf. VLDB, Stockholm, Sweeden, 1985. pp209-218

[Hil86] M.D. Hill et. al. *SPUR: A VLSI Multiprocessor Workstation.* IEEE Computer, Vol. 19, No. 11, December 1986, pp8-22

[Rei85] S. P. Reiss *PECAN: Program Development Systems that Support Multiple Views.* IEEE Trans. Soft. Eng. Vol. SE-11, No. 3, March 1985. pp276-285

[Ros87] A. Rosenthal and S. Heiler. *Querying Part Hierarchies: A Knowledge-Based Approach* Proc. 24th ACM Design Automation Conf., Miami, July 1987. pp328-334.

[Row87] L. Rowe, et. al. *A Browser for Directed Graphs.* Software – Practice and Experience, Jan 1987.

[Sch86] R.W. Scheifler, J. Gettys *The X Window System* ACM Transactions on graphics. Vol 5, No. 2, April 1986, pp 79-109

[She83] B. Sheil and L.M. Masinter (eds.) *Papers on Interlisp-D.* Xerox-PARC,CIS-5 (Revised) 1983.

[Sun86] *The SunView Programmer's Guide* Sun Microsystems, Inc. 2550 Garcia Ave., Mountain View, CA.