Active Documentation for VLSI Design

Mário J. Silva, Tzi-cker Chiueh, Randy H. Katz

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract: A new system for documentation processing in VLSI design environments is presented. The system integrates hypermedia and some CAD framework services to provide a radically new interface to CAD tools, environment and designs. This combination of technologies enables the construction of computer-based documents that are easier to create, maintain and understand. Hypermedia offers the possibility of integrating new media in design documentation and non-linear views of documents. The capability of invoking the design tools from the documentation processing system enables the re-use and direct manipulation of the design objects for the purpose of documentation. This paper describes the conceptual model for a documentation system embodying these ideas and a prototype implementation.

This project was supported in part by NSF/DARPA under contract MIP-9002962.

1 Introduction

Designers are well supplied with tools to help them create and analyze their designs. More recent work has concentrated on design management to improve the organization of the design environment. Yet, little has been done to help designers understand their designs.

In the modern CAD environment, design understanding is synonymous with design documentation. Documentation serves many purposes: specification, description, maintenance, training. Good documentation promotes reuse of design objects and processes, as well as providing a deeper understanding of the design. Inevitably, documents are a part of the design system.

In this paper, we show that a major improvement is possible if documentation takes advantage of the electronic-based media available today. Our approach uses an hypermedia system [10] as a front-end to the CAD system. Like any other hypermedia system, it provides support for the construction of multi-media presentations as a way of documenting the various aspects of systems design. Multi-media support could be useful in many ways. For instance, segments of a videotaped presentation of a design could be used as annotations to the files specifying the design artifact or the design process.

In our view, design tools can be reused as pieces of a larger documentation processing system. We extend the basic functionality of typical hypermedia systems by providing support for the concept of *active documents* [15]. An active document differs from conventional electronic-based documents by having portions that are filled-in by programs executed as it is rendered. There are many possible uses of active documentation techniques in VLSI design environments. For instance, in the production of a document, a logic simulator could be used to illustrate the timing protocol for interfacing with a library module, or a layout editor could be invoked to display cell representations, allowing readers to examine details at their will.

Having an ability to integrate documentation and design is not new. For example, this has been used with some success in programs such as *Mathematica* [16]. Mathematica *notebooks* allow the designer to sprinkle documentation among the Mathematica expressions and their outputs. The text can be expanded and hidden on demand. At least one Mathematica text book has been written as nothing more then notebooks [5]. However, Mathematica is an environment with a single tool, unlike typical VLSI design environments that support many Unix-based design tools.

Our documentation system architecture was inspired by *Intermedia* [9], a system designed for educational purposes. The Intermedia conceptual design consists of an application framework supporting hyperlinks. The framework integrates dedicated programs for manipulating new types of objects; these are linked together in the same address space. Our system is similar in organization, but the tools run in different address spaces, and we had to specify a protocol for creating hyperlinks whose anchors are objects under control of distinct programs.

From the perspective of design process management, it is also possible to collect design traces and then use them as example projects to specify a given design methodology [2]. This can also be seen as a constrained but automated way of building design documentation, where the user interacts with the example design, changing the files and re-running the tools to visualize the effects of the change. Moreover, this is documentation that can be re-used: designers use the design files as templates for the specific project at hand, modifying them when necessary.

The concept of a common front-end to the design tools has long been proposed as one of the sub-systems of a CAD framework. The system proposed here could also be used as a design flow management system of a CAD framework. However, we are exploiting a new idea, that of using an hyperlinking mechanism as the paradigm for relating pieces of data and for coordinated design tool executions, enabling construction of several forms of multi-media design documentation. As a result, we added a new viewport to the design environment, where tools, component libraries and design files can be accessed and organized in a very simple, yet powerful way.

This paper is about the conceptual model for design documentation for VLSI designs embodying the ideas presented above. These concepts are being used as part of the design of *Henry*, a prototype documentation system whose implementation is under way.

The rest of this paper is organized as follows. Section 2 describes the requirements for a documentation system, as seen from its users. In Section 3 we present the conceptual model for design documentation. Section 4 describes what tools may be used to help authors in creating documentation for designs. In Section 5 we present our prototype implementation, conclusions and future work.

2 Requirements for Active Documentation

Documentation is traditionally associated with printed paper, in the form of manuals or technical reports. Even when available on-line, the paper-based paradigm is still present. Documents are organized in some form of "pages," with their contents immutable. The only thing readers can do is to give commands to the tool used to display the document to advance to some form of a "next" page. By *active documents*, we mean those that contain other objects than static text or figures, with which readers can interact. An example of such an object could be a "figure" that instead of being interpreted by the documentation processing system as a black box containing a diagram or a table, would be the main window of a running design tool, from which the user could send commands and observe the contents of the document being changed as a result of those commands.

To find the desirable features in a system for active documentation, we begin by examining the features required by document readers and authors. Readers requirements determine what features the system should provide, while authors requirements suggest the services that support the creation of documents with these capabilities.

2.1 User's requirements

We began by looking at conventional data sheets for complex components to better understand the requirements for active documentation. To illustrate the possibilities, we examined the existing paper-based documentation from IDT [8] and asked how might active documents be used to describe exotic VLSI components, like high performance FIFOs. We found the following types of information:

General information. These are guidelines for navigating through the manual and cross reference tables to other manufacturer's "equivalent" component references.

Descriptive sections. These includes a detailed description of IDT's technology and information about its manufacturing process.

Packaging information. A reference section for the possible types of packaging used for components. This section includes equations to be used by designers for determining the physical characteristics of a given component on a specific packaging type.

Datasheets. The product description sections for each component. Datasheets present very diverse information, in fact all the information required for incorporating a chip into a larger system. This information is structured in a uniform way for each component type, such as static RAMs and FIFOs. It includes functional descriptions, tables of parameters under different operating conditions and several other types of diagrams. These are used for representing component's architecture, logic behavior, structure and timing information.

Application And Technical Notes. These are descriptions of example designs illustrating the potential applications of the components and the design steps that users should take to incorporate them in their own applications.

If the information on this databook were available as a hypermedia active document, one could benefit from the advantages of hypermedia and active documentation techniques for presenting and organizing the data and for easy navigation in several ways, as described below.

Linking mechanism for navigation. Users need a simple mechanism to jump directly to the component or component section of interest when looking into the datasheet index. They would also benefit if the packaging options available for a given component were no more than a button-press away. This requires a hyperlinking mechanism in the documentation browsing tool, so that users can easily access the datasheets or sections relevant to their objectives.

Multiple views. In order to speed-up searches, users familiar with another manufacturer's equivalent part-numbers should view those references directly. The system should use the cross-reference tables to display that information automatically, instead of having users to consult a document section presenting those tables.

Multi-media. The system should support multi-media representations. For instance, to explain a CMOS manufacturing process, video and animations could be used to illustrate the various steps.

Queries to locate information. In many cases, users know some of the properties required for the component they are looking for, but do not know if that component exists. To facilitate searches, they should have a query facility to look for specific components by giving a list of required attributes. Using active documentation techniques, the existing databases for storing information about components could be made accessible directly from the new electronic databook.

Actual data, not formulas or parametric curves. Instead of presenting the user with a set of formulas to compute the physical properties of the component on the package, those properties should be computed by the system using the available information for the component in the selected packaging option. This way, it would become easier to exploit design trade-offs. This suggests the need for a *spreadsheet* facility that document writers could use to specify packaging properties. The system could invoke this facility to compute the actual values for a component to fill-in a table with the packaging data.

Datasheets are an example where dynamic invocation of design tools, to present the contained information, would be most useful. Along with the capabilities above described, direct use of the CAD tools presents new possibilities. Instead of timing diagrams, it would be interesting to present users with a diagram produced by the CAD system "on the fly". Instead of generic characteristics or characteristics at specific operating conditions, the user could access directly the same information at the conditions of operation required by the application at hand.

Ability to play with document data. Application Notes are example designs, built for the purpose of illustrating for users the interconnections and design considerations for using the component within a custom system.

Component users would benefit if the design files used in the application note were made available as part of the documentation. Moreover, if the application note included the scripts used for building the design, users could "replay" the construction of the design, and subsequently re-use the same design files and design methodology as a template. With application notes based on the electronic media, they may be reinterpreted as documents with informative text for the example design files and tools, associated with a navigational aid for helping other designers understand those files and the tasks needed to build the design.

2.2 Author's Requirements

After enumerating the requirements from the users, we could derive what services the system should provide to those writing hypermedia active documents describing designs.

Synchronization of CAD tools execution. A design description must be composed from information in independent windows, each generated by a distinct tool. The author needs a capability to synchronize their execution. For instance, one might want to explain in a text window the composition and functionality of each of the main blocks of a microprocessor while the layout and schematic of each block are displayed in graphic windows aside. When the reader scrolls the text window to the description of another block, the graphic windows should change their contents to display the corresponding representations.

This requires the ability to send commands from the tool presenting the textual description to the CAD tools used to display the graphic design information and have them interpreted as if they were given interactively by a user.

Convert screens into portions of a document. In some situations it is not possible to reproduce the interactions used to reach a given design state. This may be because some of the tools used are not prepared to interact with the design navigation system and it is impossible to make them replay the actions taken. Another plausible reason is that they may take too long and it is not desirable to have readers waiting too much for a requested document "page." In these situations, to offer some aid to the document writer willing to illustrate a given design step, the system should provide a capability to dump the contents of windows or screens at a given time, to have them redisplayed later for inclusion as part of a document. Even though this solution does not offer the reader the possibility of interacting with the tools from the document, the screen dump could still be annotated and used as an illustration to a textual description describing that design step. Another possibility is the recording of the messages exchanged between the window system server and the tools during that design step. The recording could then be replayed later to show the designer's commands sent to the tools. Both capabilities are already available on current workstations software.

Produce documents from a template. The most obvious way to produce electronic documentation is by giving authors the metaphor that they are adding "pages" incrementally to an existing document. However, data books are a collection of data sheets organized in a common way. Documentation writers job would be greatly simplified if they could generate documents by filling-in a predefined set of "pages," each corresponding to a policy imposed section of a document, for example the template for a data sheet.

Control of what is active in a window. Authors also need a mechanism to control what in a design may be modified by the document readers. It may be desirable not to allow users to modify some parts of the document even though they are given a tool which allows them to change the design files displayed. In other situations, parts of a document are intended to be modified by the readers in order to further exploit some characteristics not explained in a document. As an example, consider a document where the electrical characteristics of a circuit are presented by a simulator. The circuit model is stored in a file, and the simulator uses that model to evaluate the timing diagrams that will be displayed. A normal requirement in these situations is that readers should be able to change the specifications of the circuit loads or input waveforms, while not allowing them to change the remaining of the model file.

In system design, the world can not be divided clearly into document readers and authors. System maintainers are a class of users who may be both readers and writers during a session with the documentation system. Protection has to be established at a finer level of granularity then the level of an entire document. This calls for the definition of a concept of protection for document manipulation that can address these needs.

Internal documentation and Databooks. In many situations the same design will require several classes of documents. The documentation available internally for design maintenance of VLSI components is generally not the same as that published in datasheets. In active documents, the tools available in house for producing the designs may not be available for those who use the documentation. This implies that authors should have the capability to define classes of readers, organized according to the models, type and level of detail of information they will have access, and the tools they will have to process the documentation. Then, different versions of the same document would be produced for each class. It is still possible to provide active documentation services when design tools are not included with the documentation processing system. One may use public domain design tools or tools from a CAD vendor supplying the reader's site. Alternatively, standard representation formats, such as VHDL models, that can be processed by tools from several possible vendors could be used.

3 The Model

Our model for documentation is based upon hypermedia. Our system inherits from hypermedia systems the *linking capability*, where a user has the flexibility of selecting the next piece of information to access at the time of reading. However, our architecture does not follow conventional hypermedia systems, where a set of programs, each able to present a specific type of media, are bound together with an hyperlinking mechanism. The architecture of the documentation system proposed here more closely resembles a traditional CAD framework, where several tools can be invoked from a coordinating tool, to build the screens required to perform a given task. Our design navigation system can thus be seen as a methodology management system of a CAD framework using hypermedia links to organize the data and to reuse tools and design processes.

3.1 Data Structure and Concepts

The conceptual model shares many similarities with the model of HIP [1]. As in any hypertext model, the basis for our documents is a network where the nodes (or *contents*) correspond to data, and the arcs (or *links*) identify which nodes may be visited from a node when traversing the network. Links have one *anchor*, representing the origin of the link, which may be any object selectable from the user interface of a tool displaying a node. A document is an entity that is processed by a specific tool, called the *navigator*, which corresponds to the hypermedia system front-end. The *navigator* has its own data structures and files totally independent of the files and tools of the design framework. These data structures represent, among other things, the linking information and sequences of nodes for traversing the network, defined by the document writers. Readers may then choose from a set of suggested reading orders, called *guided tours*, or follow their own.

Figure 1 shows a typical screen. Nodes are displayed according to a scrolling-view presentation model V, meaning that each node is displayed in a scrollable window. Documents are organized in *frames*, each frame describing the position on the screen of a set of windows. Each of these windows, corresponds to the display area of a *contents* in the hypertext network. Document writers have the ability to control what part of a *contents* is visible on a scrollable window when the window is displayed, and document readers may interact with individual windows of the frame. However, a frame in *Henry* has a very distinctive appearance. *Henry* does not organize all windows in a frame as descendents of a common top level window in the window system hierarchy, nor does have all windows in a frame controlled from a single tool. *Henry* represents a frame as a set of top level windows. Each window may be run from a different process, but from the control panels of the navigation tool they are seen as logically grouped and they may be manipulated as a single entity. When displaying a document, the request of a new frame corresponds to adding a set of windows to the desktop, in relative positions determined by the document creator.

Figure 2 shows the interface for manipulating documents. There is a control panel, labeled "henry" in the figure, from which users give commands to access documents and obtain information on how to use the system. For each document opened or created, a new control panel, called a document control panel, is displayed. From this panel, users may give commands to navigate within the document or manipulate its frames. For navigation, a user may choose from a set of possible frames what will be viewed next via the ***Select*** pull-down menu. He may also follow directly to the current default next frame via the ***Next*** button. The default next frame is determined by the current *path*, which corresponds to one of the possible guided tours defined for the document. From the document control panel, users may also manipulate the frames (using the pull-down menu labeled ***View***) and invoke other menus to display navigational aids for the document. With a single command, a user can dismiss all the windows of a frame previously rendered or bring the same frame again to the top of the desktop. A third control panel (not shown on Figure 2), invoked from the document control panel, allows authors to define or change the contents of each frame. They may specify what windows are

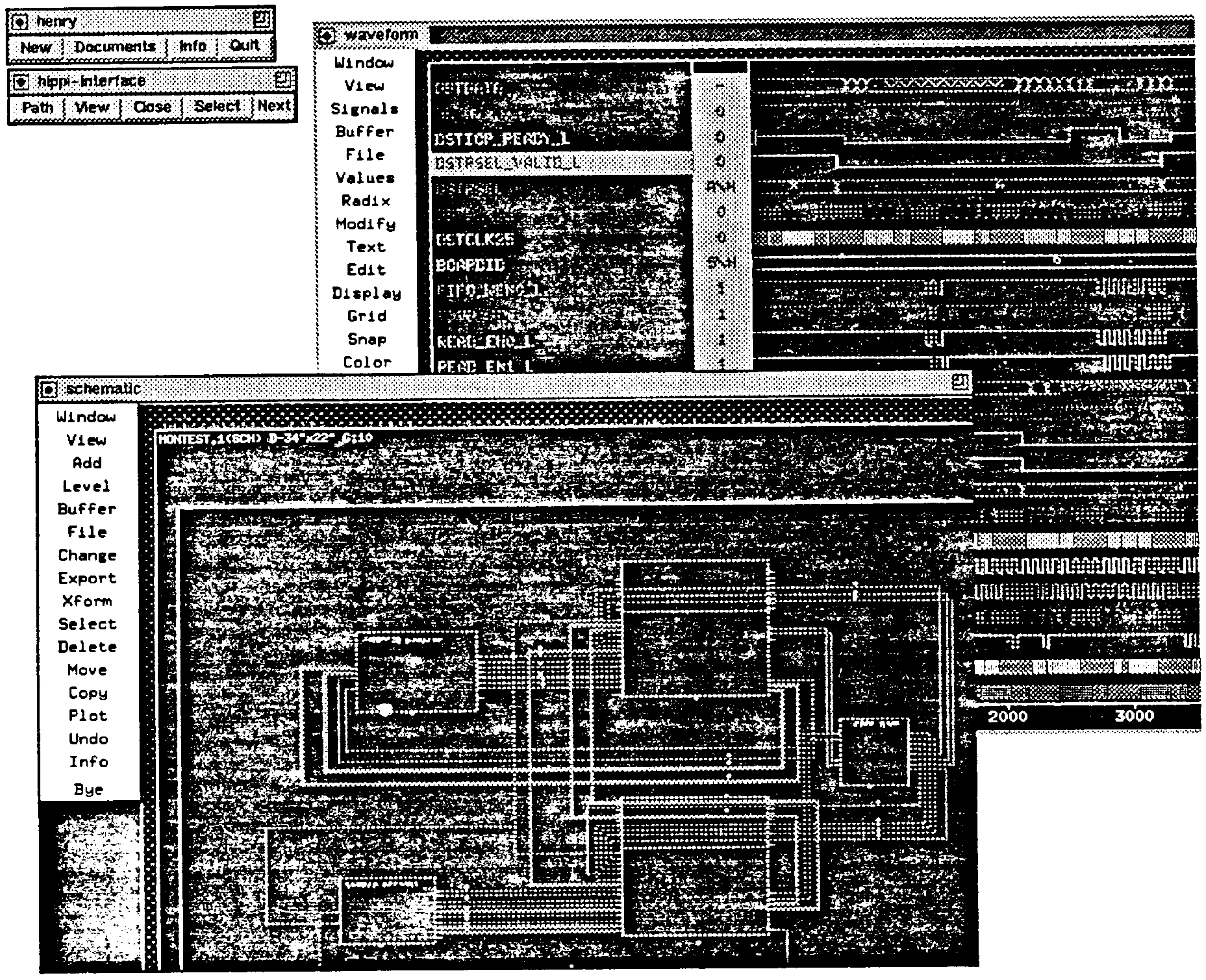


FIGURE 1. Representation of a document in terms of the model. The *navigator* starts by displaying its control panel (top left), from which the user has opened the "hippi-interface" *document*. This in turn triggered the creation of another control panel, from which the navigation on the document may be controlled. A *frame*, consisting of the two windows on the right is being displayed. These correspond to two independent *contents* in the hypermedia network defining the document. Objects in each tool's space (such as the selected signal in the "waveform" window) may be defined as *anchors* which, upon user's activation, will trigger the execution of *actions* that will run by on the *navigator's* space.

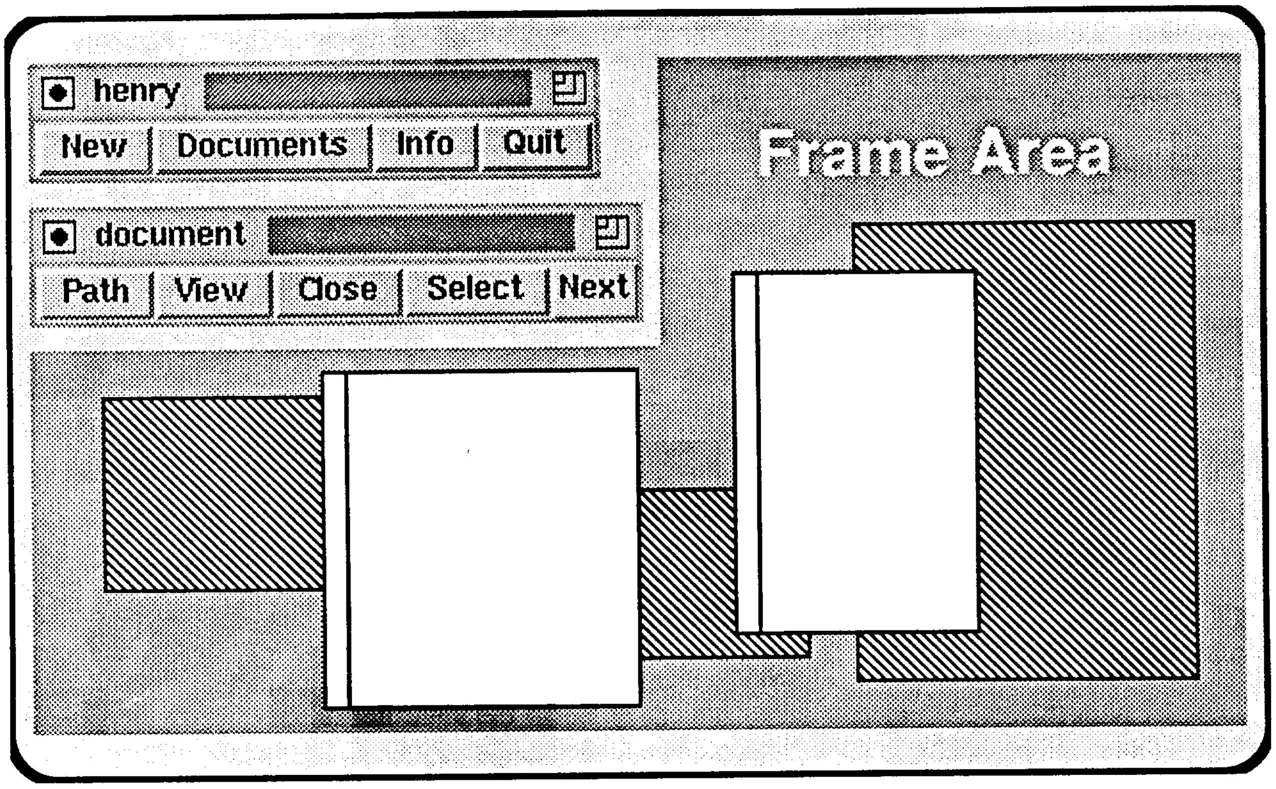


FIGURE 2. The control panels available to a user reading a document. The top panel is the system control panel, from which the user obtains help and opens existing documents. A document control panel (the panel below), is created to control each opened document. From it, users give commands to manipulate the frames of the document. Each frame has a set of top level windows. When a frame is invoked, its windows are displayed on top of the windows of the frames previously rendered.

displayed in a frame, their geometry relative to the screen and what program has to be run to display their associated *contents*.

Each *contents* corresponds to a node in the hypertext network, and its rendition to the screen is controlled by a tool, independent of the *navigator*, but with the capability to communicate with it by a common RPC[1] protocol. This protocol defines a set of commands that can be passed between the *navigator* and the tools, used to control their simultaneous execution. For instance, the protocol allows for the definition of any object in a tool's domain as an *anchor*. In the frame representation on Figure 1, the name of the signal "DSTR_SEL_VALID_L" on the window "waveform" could be defined as

1. RPC — remote procedure call

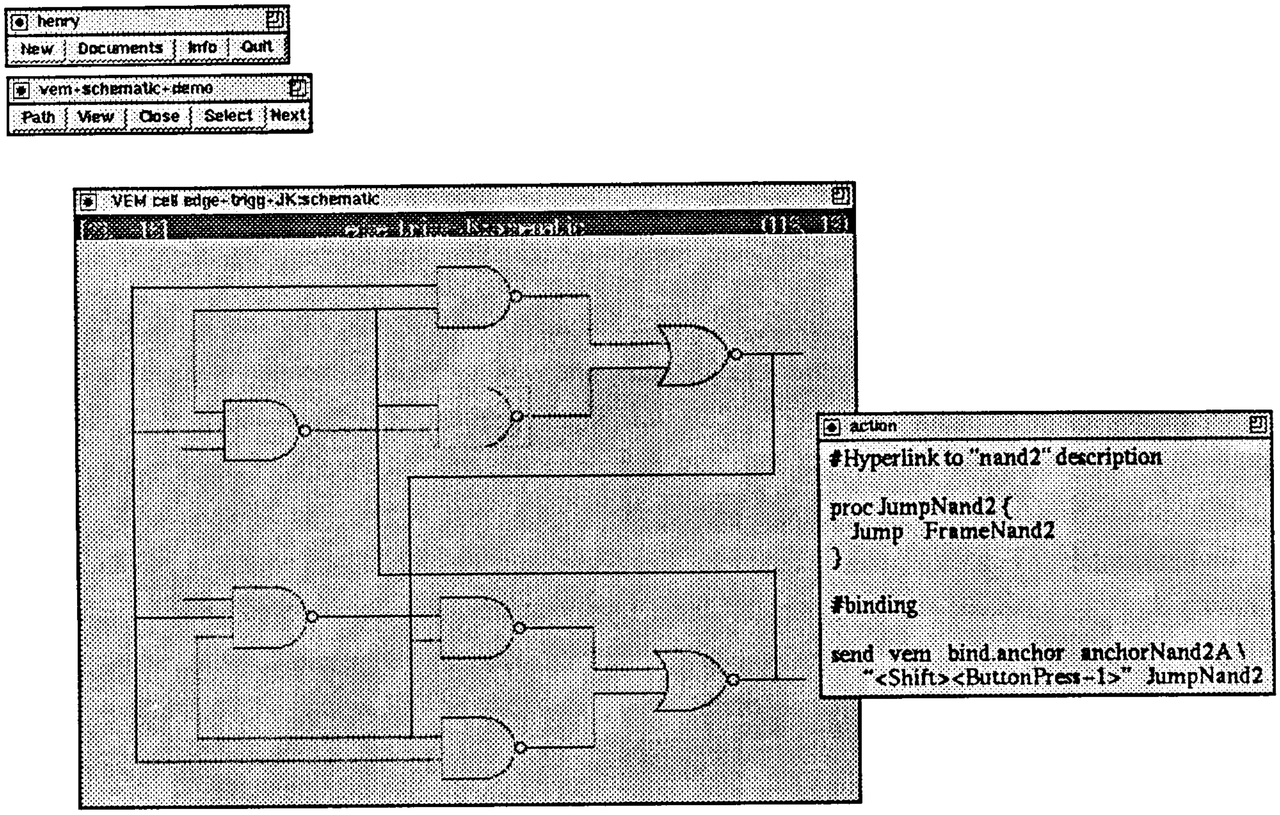
an anchor, meaning that upon a button press event, the *navigator* would be notified to display some other frame or to display other contents on the other windows of the same structure. Another *anchor* could be the graphic object representing the interface of some component in the "schematic" window. This *anchor* may be defined to behave as a button, which will render another frame with the schematic of the component shown as a black box together with a description of its operation. When an event occurs within the context of one *anchor*, a notification is sent back to the navigation tool, which in turn may have an *action* associated with it. *Actions* are programs written in a system extension language, allowing document writers to implement several possible functions, such as invoking another tool to display another *anchor*, which in the hypertext world corresponds to a *link*.

The concepts of the model are elaborated below.

Anchor. From the hypermedia system point of view, an *anchor* is a reference to some object recognized by the *tool* presenting a *contents* object. An anchor description consists in two parts, one to be interpreted by the tool running on the contents window, and another that will be interpreted by the *navigator* itself.

The part interpreted by the tool contains a definition of the identifier of the object associated with the *anchor* in its tool address space plus a description of what events should be sent to the *navigator* upon user's interaction with the *anchor*. In a *contents* consisting only of textual information, an *anchor* attached to a word in the text could consist of the order of the first character of the word in the text plus its length; in a *contents* displaying the layout of a circuit, an *anchor* could be defined as the identifier on the design database for an object such as a "port" or an "instance" of a given "cell."

The part interpreted by the *navigator* consists in a script of commands, called an *action*, to be run in response to each event sent by the tool. The scripts are written in an extension language interpreted by the *navigator*. The extension language offers the capability to start tools and send commands to other running tools to perform specific actions. Figure 3 presents an illustration of this mechanism, showing how hyperlinks are implemented with the ***Jump*** command. Another command, ***Display***, is used to tell a given tool to display an *anchor*, meaning the tool will scroll the associated window, making the *anchor* visible. This way, *anchors* may be used both to synchronize presentations where two or more independent tools are used to display related pieces of information, and to implement the linking mechanism.



```
#Hyperlink to "nand2" description

proc JumpNand2 {
  Jump  FrameNand2
}

#binding

send vem bind.anchor anchorNand2A \
   "<Shift><ButtonPress-1>" JumpNand2
```

FIGURE 3. How *anchors* work. On this frame, the navigation tool and the schematic editor have commonly *defined* the shaded *"nand2"* instance as an anchor When a user selects the anchor (through a mouse click over the area of the symbol corresponding to the anchor) the *action* associated with the anchor is run. In the example, it contains a single command to *Jump* to another frame, implementing a hyperlink.The two programs communicate via RPC.

Contents. The concept of a node is represented by the type *contents*. A *contents* may be seen as the encapsulation of a file and the tool used to manipulate it. It also contains the set of *anchors* defined for the contents.

Frame. A *frame* describes a set of *contents* plus a user-interface to perform actions on its elements. The description includes the layout of the *contents* displayed on the screen when the frame is rendered and the position and semantics of a set of *buttons*, used for bringing pull-down menus to edit the document, select the default path, or the next frame to be invoked.

Button. A *button* describes one action that the user may invoke. It consists of a graphical representation and a*n action*. Buttons are used in frames to allow for navigation without having to interact with the tools that may be invoked by the navigation system. A common problem in hypertext systems is that users tend to find themselves "lost in a hyperspace of data "[10] when they do not have a good visualization of the structure of the document. A set of buttons common to all frames in a document, may be used to offer standard aids for navigation such as guided tours or escape buttons. Buttons can also be used for implementation of *links* in an alternative way to *anchors* defined in *contents*. This is specially useful to allow the use of foreign tools[1] in active documents.

Path. A *path* is the data type that provides support for guided-tours inside the hypermedia system. Unlike most systems that define a *path* just as a sequence of nodes, our concept of a path is a mapping from each *frame* to a destination *frame* and a set of *anchors* defined within it. This corresponds to rendering the destination frame in such a way that all the anchors are visible in their contents windows, when a pre-defined button, the *next* button, is activated. The user of the system, after selecting one of *paths* defined for the document, follows a guided tour by pressing the *next* button, corresponding to the default link. This will advance the traversal to the destination frame defined in the *path* for the frame being displayed. He may jump out, forward or backwards of the path freely, but has always accessible the possibility of following the established default path once he steps into a frame in the path.

Document. A *document* is the top level data type. It is described by a *name*, a set of *paths* and a set of *frames*.

3.2 Operations

A document is stored as a single data file, called a *web* [9], defining the structure of the document in terms of the data types above described. The basic program in the *Henry* system is the *navigator*, that presents *documents* as defined in their associated *web* files and allows users to add new *frames* and *links* interactively. The *navigator* has a command interpreter and a graphical user interface based on the Tcl language and the Tk toolkit [11] [12].

Tcl is an embeddable command language, extensible by applications. One Tcl command, *send*, allows any application with a Tcl interpreter to dispatch a command to be executed on any other application running on the same screen. Tk is an X window system [13] toolkit, whose interaction objects communicate with applications through commands built as an extension to the Tcl language.

1. those that are not be modified to implement the RPC protocol required for creating anchors.

Together, Tcl and Tk form a flexible and dynamic user interface description language. Our extension offers additional support for handling active documents.

A *web* file is itself a script containing commands defined as an extension of the Tcl language. It is beyond the scope of this paper a formal definition of the document description language. We have considered the adoption of a standard document description language such as SGML [14], but these do not support the concept of active documentation. Although a *web* file may be created and edited using a text editor, it is not intended to be manipulated that way. *Webs* are parsed by the *navigator* to initialize its data structures. Authors may then modify the data structures interactively, and they will be written back to the *web* files in the format of the design definition language upon exit.

Anchor definition. Anchors are defined using the select, cut and paste mechanism available in modern graphic user interfaces [4]. A selection is basically a buffer with a protocol for access by several concurrent programs. First, users start by selecting the object they want to define as an anchor in the tool displaying the anchor interface. The result of this is that the tool places some data in the selection buffer that will define uniquely the object chosen to be associated with an anchor. Then the user instructs the *navigator* to define an *anchor.* The *navigator* will use what is currently in the selection buffer as the data defining the object previously put in the selection by the user. The data passed to the *navigator* is a persistent definition of an *anchor* representation, meaning that it may be passed back to the tool in another run as part of a command to recreate the anchor in the tool's address space.

Link Creation. In order to execute a jump to another frame, the system has to perform the following operations:

- Invoke the tools associated with each *contents* in the *frame.*
- Instruct the tools to create their windows on the desired locations.
- Command each tool to "scroll" to display the required *anchor* on the associated window.

To define a *link*, we need to define previously at least one *anchor* for its origin. The *anchor* will have attached the name of an *action* to be executed when a user interacts with it, for example by pressing a button.

Actions. In our prototype implementation, the scripts that describe the *actions* to be taken when an *anchor* is selected are procedures in the Tcl language. In these procedures, the Tcl commands defined by the Tk toolkit may be used, making it a very powerful way for defining user interactions. One of the commands available in the scripts is the ***Jump*** command, that provides the hyperlinking capability. ***Jump*** takes as arguments the name of the destination *frame* and optional pairs with the name of a *contents* and an *anchor*, to specify what will be displayed in each of the corresponding windows.

Hierarchies of documents. One simple way to support hierarchies of documents is to have a *contents*, whose tool is another instance of the *navigator*, configured to process another document upon start-up. This has the limitation that gives the user multiple contexts in a single session: each invocation of the navigator would create additional control panels to manipulate the documents accessed this way, leading to a poor user interface. The solution is to merge dynamically the *web*s of other documents upon user demand. For this reason, we provide an additional option in the ***Jump*** command, indicating the name of the document the destination *frame* belongs to. The document will then be merged if it has not been accessed before.

4 Document Building Toolkit

For hypertext-based documentation systems, no unique paradigm for writing documents exists. The bottom-up and the top-down processes are equally used [10]. Generic hypertext systems have only limited aids for document builders, such as document consistency checkers to verify that there are no dead-end nodes or browsers for the document structure. Besides the usual aids in hypertext-based systems, *Henry* offers a set of new tools that may be used by document writers to speed-up the process of constructing design documents. Hodges and Sasnett [7] proposed the idea of "plastic editors," customizable editors that document writers could include in their multimedia presentations. In our system, the CAD tools are also viewed as customizable editors that designers may use to document their designs. We take advantage of being restricted to an hypermedia-based application dedicated to hardware documentation, with the additional capability of having the possibility of reusing the tools and services of a CAD Framework and the design data for building documentation. For instance, in a top down process for writing documentation, one could think of a tool to generate automatically a *document* having a *frame* for each component in a design and *links* reflecting the hierarchy.

Henry offers document writers the possibility of automatically adding nodes with a pre-defined functionality to an active document. Framework services and tools can be encapsulated in *frames*, which can then be instantiated in a document being assembled. This way, several kinds of information about a design such as its history or its structure can be incorporated into documents as active nodes with minimum effort. Most tools to manipulate frames in the document building toolkit in this class are simple filters that convert a *web* into another file in the same format, adding or modifying *frames* in the process.

Another class of tools in the toolkit is that of those that are modifications of existing browsing tools and that can be put in a document to help navigation. For instance, in the VLSI environment, a layout browser could be modified to display the *frame* containing the documentation about one component when a user presses a button over the area defined by its protection frame. In fact, the interface between *Henry* and the tools works in both ways, giving the system a dual purpose: it can be used both as a documentation system for designs and as the help system for the design tools.

Finally existing tools to display multimedia information may be also employed in *Henry*. For instance, a portion of a videotaped presentation describing the behavior of a given component, could be made available from a frame which would be linked to the file containing its representation.

5 Status, Conclusions and Future Work

Henry is being implemented as an assembly of several subsystems. The framework used is the Octtools[6]. We are modifying some tools to incorporate a Tcl interpreter to accept commands for defining *anchors* and sending events to the *navigator*. The current version of the *navigator* is a Tcl/Tk based application under development, that will provide minimum functionality to test the conceptual model under development. We are also working on the integration of the FrameMaker publishing software within *Henry*, communicating via FrameMaker's RPC interface [3]. Figure 4 gives a pictorial representation of the general architecture of the documentation system, in terms of files and processes, showing how the *navigator* is integrated with the publishing system and the CAD framework above described.

We are also evaluating current multimedia tools and hypermedia systems, offering support for the manipulation of video data, that could be tailored to allow the implementation of the model here described, thus offering the possibility of building multimedia presentations embedding the VLSI design tools we are currently using and offering the possibility of using the document building toolkit under development.

We presented a model for documentation of hardware designs, using the idea of active documentation and based on multimedia representations. We also proposed the notion of a hardware document building toolkit, describing the set of tools available to help the tasks of writing hardware

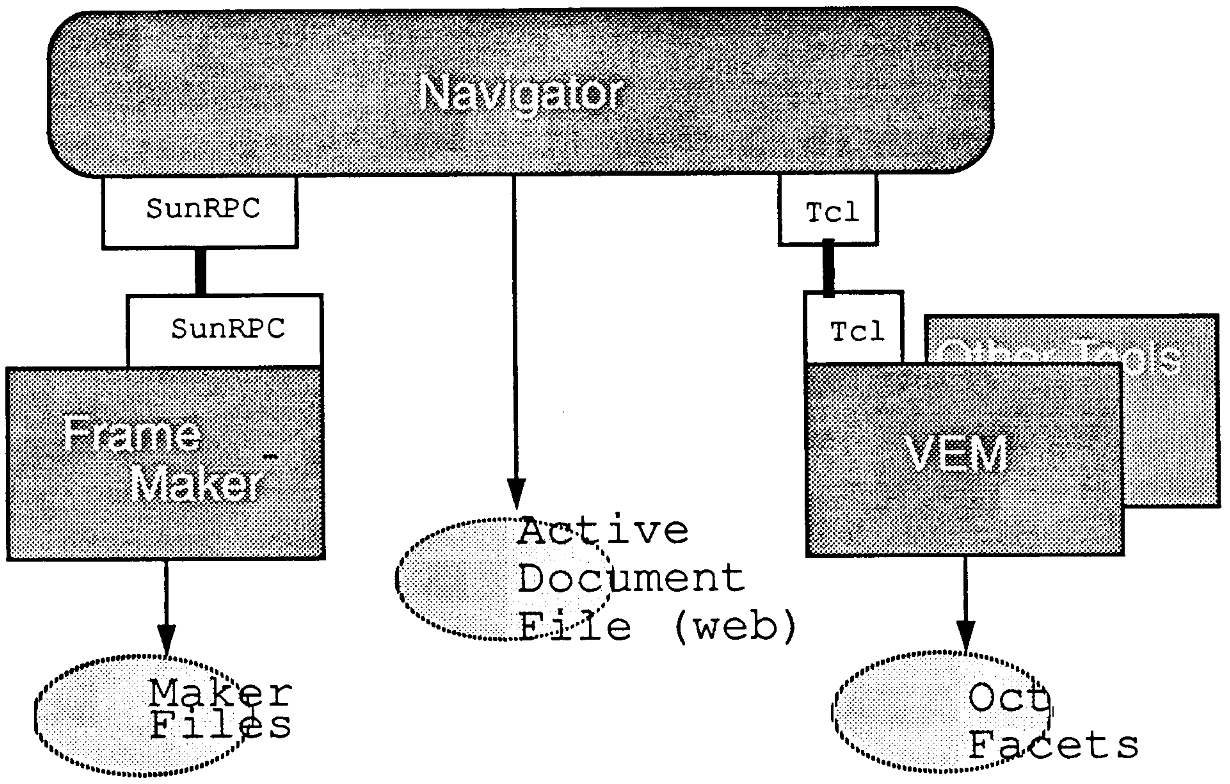


FIGURE 4. The architecture of *Henry*. This figure shows how the navigator communicates with FrameMaker via the Sun RPC interface and with a modified version of VEM (the octtools editor) via Tcl.

documentation. This model and the toolkit to help writing VLSI designs documentation are being designed as part of the *Henry* documentation system.

Tcl/Tk have proven to be extremely powerful for the development of this application. First, because of its extendable language and embedded communication mechanism provide an invaluable support for the implementation of *Henry*, which requires a generic mechanism for transferring control commands between tools. Secondly, with Tcl/Tk it is a matter of minutes to develop a small program supporting simple user interactions to pass commands to other tools. These are in many situations required as a simple means to invoke an extra tool to a *frame* or to provide an hyperlinking button when writing active documents.

We are developing *Henry* as a vehicle to research on how multimedia technology can be employed in CAD systems to help designers, specially in system design documentation. We intend to add support for real-time video data capture and investigate how multimedia information such as videotapes of design reviews and design meetings could be used to significantly reduce the time dedicated to documentation while improving its quality.

6 References

[1] Beverly S. Becker and Lawrence A. Rowe. *HIP: A Hypermedia Extension of the Picasso Application Framework*, December 1990. University of California, Berkeley. Memorandum no. UCB/ERL M90/121.

[2] Andrea Casotto. *Automatic Design Management Using Traces.* Ph.D. thesis, University of California, Berkeley, March 1991. Memorandum no. UCB/ERL M91/22.

[3] Frame Technology Corporation, 1010 Rincon Circle, San Jose, California 95131. Integrating Applications with FrameMaker, Part Number 41-00327-00, 1989.

[4] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Mass., 1984.

[5] Theodore W. Gray and Jerry Glynn. *Exploring Mathematics with Mathematica.* The Advanced Book Program. Addison-Wesley Pub. Co., 1991.

[6] D. S. Harrison, P. Moore, R. L. Spickelmier, and A. R. Newton. *Data Management and Graphics Editing in the Berkeley Design Environment.* In Proceedings of the 1986 IEEE International Conference on Computer-Aided Design, pages 24–27, 1986.

[7] Matthew E. Hodges and Russel M. Sasnett. *Plastic Editors for Multimedia Documents.* In Proceedings of the 1991 Summer USENIX Conference, pages 463–473, 1991.

[8] Integrated Device Technology, Inc., 3236 Scott Boulevard, Santa Clara, California, 95054. 1990-91 Specialized Memories Data Book, 1990.

[9] Norman Meyrowitz. *Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework.* In OOPSLA'86 — Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 186—201. ACM, September 1986.

[10] Jakob Nielsen. Hypertext and Hypermedia. Academic Press, Inc., 1990.

[11] John K. Ousterhout. *An X11 Toolkit Based on the TCL Language.* In Proceedings of the 1991 Winter USENIX Conference, 1990.

[12] John K. Ousterhout. *TCL: an Embeddable Command Language.* In Proceedings of the 1990 Winter USENIX Conference, 1991.

[13] Robert Scheifler and James Gettys. X Window System, Digital Press, 2nd edition, 1990.

[14] SGML. ISO/IEC 879-1986 Standard.

[15] Douglas B. Terry and Donald G. Baker. *Active Tioga Documents.* Xerox Corporation Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304. Technical Report CSL – 90 – 6 June 1990 [P90 – 00113].

[16] Stephen Wolfram. Mathematica: A System for Doing Mathematics by Computer. Addison-Wesley Pub. Co., 2nd edition, 1991.