HyperWeb: A Framework for Hypermedia-Based Environments

James C. Ferrans, David W. Hurst, Michael A. Sennett,
Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang

Vista Technologies, Inc.
1100 Woodfield Road, suite 108
Schaumburg, Illinois 60173-5121

ABSTRACT

Software productivity and quality will increase as we improve our model of software and develop tools to support that model. Development environments must take into account that software is more than source; that it is more than text; and that it forms a highly interconnected web of information. Because more time is spent understanding and maintaining software than creating it, environments should strongly support browsing and reading. Finally, environments must be easy to customize.

In this paper we present HyperWeb™, a framework that supports the construction of hypermedia-based software development environments having this richer view of software. It coordinates the activities of an integrated set of tools through a message server, uses an object-oriented database to store software artifacts, and supports hypermedia linking of these software artifacts. It is built around an interpreter for a general purpose scripting language, allowing for very flexible customization and environment building. We also describe our experience in using it to build an environment that supports software design, development and maintenance on Unix™. Its primary features include support for document linking, source code annotation and restructuring, and modification request tracking. It is being used and evaluated internally and at several external sites.

1.0 INTRODUCTION

HyperWeb is a general framework that accesses a database of software artifacts at the back end, and coordinates a set of integrated development tools at the front end. The system is built around an interpretive scripting language. It implements hypermedia, and is open and customizable. Software artifacts are stored in a distributed Object Management System (OMS) that meets the Portable Common Tools Environment (PCTE) standard. The HyperWeb concept and design were developed at Vista Technologies [Ferr89] starting in 1989 and implementation has proceeded since.

Our most recent efforts have been focused on using HyperWeb as the basis for an environment which supports our insights into software. The environment we have built supports general Unix software development while using HyperWeb's hypermedia capabilities to facilitate linking between software artifacts, restructuring and annotation of source code, and tracking of modification requests.

In the remainder of this section we discuss the background of the HyperWeb project: what insights into software development motivated us, and what related work influenced our approach. In the next section, we describe HyperWeb's integrating framework. Section 3 covers the standard tools integrated with HyperWeb and our experiences integrating "external" tools. Section 4 describes an environment built on the HyperWeb framework to support Unix software development and maintenance.

1.1 Software and Software Development

Several key insights about software development have guided us in this project. First, *software is more than source*: artifacts of the development process might also include rough ideas, requirements, designs, reviews, inspections, and problem reports. *Software is also more than text*: design diagrams, screen images, and even audio and video annotations can be considered software.

Moreover, *software is not a collection of isolated objects*: software artifacts form a highly interconnected, non-linear web of information about the complete software system. Requirements are met by design elements, design elements are realized in code, and modification requests relate to the objects that must be modified. Today, developers must remember many of these relationships; tools that reduce this cognitive load would help immensely.

Software is read more than modified. Developers spend more time reading software than writing it; there is a great benefit to anything that makes software more readable and understandable.

Finally, *there is no one right way to develop and maintain software*. Each enterprise, development group, and even programmer has different problems and different ways to handle them: environments and tools must be readily customizable.

1.2 Related Work

Clearly, CASE tools should support the insights into software and software development described above to improve productivity and quality, but how? HyperWeb answers this question by bringing together such diverse technologies as hypermedia, literate programming, and tool integration frameworks.

Hypermedia is a data structuring and user interface technology popularized by Xerox's NoteCards [Hala87] and the Macintosh HyperCard system. These two systems demonstrated the power of building a hypertext system around a scripting language. The idea of applying hypermedia to software development has intrigued many researchers; it seems to be a natural way to capture the relationships between the various components that comprise a software system. Textronix's Neptune [Deli86] was an early attempt to utilize hypermedia for CASE and CAD applications. Neptune had a variety of browsers which helped address the "lost in hyperspace" problem and supported versioning, two crucial characteristics for hypermedia applications. Amdahl's HyperCASE project [Cybu92], which emphasizes design and project tracking, is a more recent effort to combine CASE and hypertext. However, neither of these systems provides a well-defined mechanism for integrating existing software development tools. The IBIS system [Ritt73] and MCC's DesignJournal [Conk87] have shown how hypertext can be used to track designs and to capture information and discussion supporting design decisions.

Literate programming is Donald Knuth's approach to software development [Knut84], Its goal is to organize software for humans, not the computer. In WEB, Knuth's implementation of a literate programming system, you build software in sections. Each section has code and optional documentation in a text formatting language. The "weave" program prints a beautiful document containing these sections in the order you wish to present them to the reader. The "tangle" program extracts the code and assembles it in the order required by the compiler. One section can be decomposed into others, to model the software's natural design hierarchy, which is otherwise lost when the design is translated to code. Although proven in developing large programs, WEB has limitations. You work with a hardcopy of the woven output, which gets out of date, and does not match the representation you edit. You follow interconnections between sections by leafing through them as you would with a book. These problems would not occur in a hypermedia implementation of literate programming. You would use "what-you-see-is-what-you-get" editors, and follow connections with mouse clicks. You could have a rich variety of material linked to your source. While we have not yet implemented a full literate programming environment, HyperWeb can support them, and its design has been strongly influenced by this approach to structuring software.

Finally, several developments in the area of *integration frameworks* are significant. Steve Reiss at Brown University with the Field environment [Reis90] laid the groundwork for the message-passing approach to tool integration, introducing the concept of a message server as the coordinator of tool interaction. Hewlett-Packard's SoftBench [HP89] was the first commercial product to utilize this technology. The Portable Common Tools Environment (PCTE) is an initiative of the European Strategic Programme for Research in Information Technology (ESPRIT) and is a basis for building software engineering environments [Boud88]. ESPRIT recognized that its advanced programming environment projects all needed to share a common substrate of tools. Beginning in the mid-1980's, it funded the development of the PCTE interface specification. In 1990, the European Computer Manufacturer's Association (ECMA) adopted an advanced version of the PCTE specification as a standard [ECMA90]. Several implementations have been started, of which GIE Emeraude's is the most complete.

2.0 HYPERWEB'S INTEGRATION FRAMEWORK

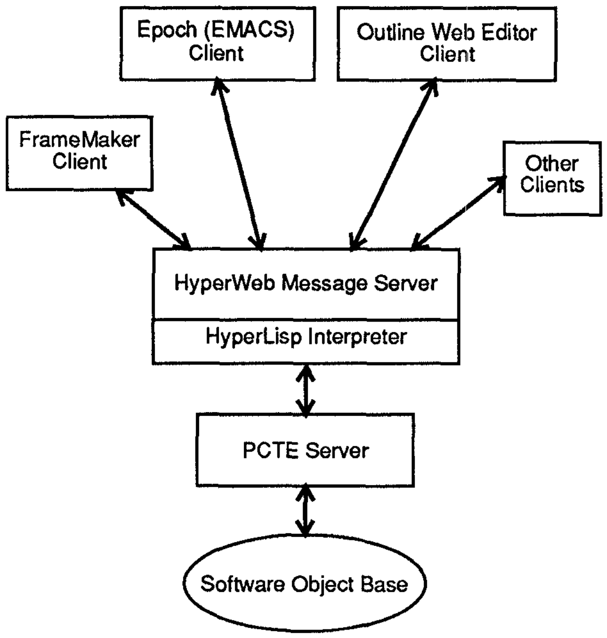
Figure 1 shows the architecture of the HyperWeb framework. The HyperWeb server provides the *control integration* for a single user's session. The software development tools being used by that developer are all clients of the HyperWeb server, sending messages to it and receiving messages from it. The HyperWeb server communicates with a PCTE server which accesses a software object base, providing *data integration*. The HyperWeb server also provides mechanisms that support *presentation integration* in environments built using HyperWeb.

The HyperWeb framework coordinates tool activities through message passing, as does the Field environment [Reis90], and Hewlett-Packard's SoftBench [HP89]. But HyperWeb extends this approach in three significant ways:

- It uses a PCTE-compliant OMS to store software development artifacts of all types. PCTE supports communal work through data distribution, views, transactions, locking, composite versioning, and access control.

- HyperWeb fully supports hypermedia. Each hypermedia node is represented as an object in the object base, and each hyperlink is stored as a relationship between two objects[1]. A hyperlink can go from an anchor point in its origin node to an anchor in its destination node. PCTE schemas constrain the types of links that can leave or enter a node: this imposes a regularity on the object base that reduces the disorientation of unrestricted hyperlinking.
- The HyperWeb server is built around an interpreter for a dialect of Lisp. This gives a high degree of customizability to the environment builder. Application-specific logic is executed in the server rather than by individual tools, simplifying customization and tool integration.

FIGURE 1. HyperWeb framework architecture.



2.1 Control Integration

In approaching control integration, our primary goal was to achieve expressive power and flexibility. For this, we chose to build the control integration around an interpreter for a hypermedia scripting language. The scripting language would be a flexible glue for constructing environments. A scripting language approach has been successful in such products as Apple's HyperCard™, GNU Emacs, Interleaf™, and AutoCAD™. We chose to base the message server around a Lisp interpreter providing compatibility features with Common Lisp and supporting a style of object-oriented programming. Our dialect of Lisp is called HyperLisp.

To the Lisp interpreter we added a general inter-process communication mechanism, based on TCP/IP sockets. This allows client tools on both local and remote hosts to communicate with the server. Messages are ASCII text, for ease of communication across machines with different architectures. There are HyperLisp primitives for sending messages from the server to clients. Messages coming into the server from clients must be Lisp expressions. A *call message* blocks the sending process until the receiver replies, and an *event message* requires no reply from the receiver.

A crucial aspect of our approach is that every client process connected to the server has a HyperLisp *client object* in the server that acts as its sole agent. All communication to the client process is filtered through its client object, and all communication from the client process likewise comes through its client object. Client objects implement methods that meet a layered set of standard protocols. There is one basic protocol that all clients must meet, another protocol for those clients which are used to edit a node's contents, and so on. These protocols are designed so that once a client tool is integrated, it can be used in any number of environments; all environment-specific logic resides outside the client tool and its client object agent. The protocols also ensure that new clients can be integrated without affecting other integrated clients. Client objects are instances of HyperLisp classes, and inheritance is used to simplify tool integration. For example, if a new editor is integrated, its client object will inherit from the abstract superclass *NodeEditor.*

When a client connects to the server, its client object is created and placed in the *client registry.* This list is the basis for broadcast messaging. Without the registry, only point-to-point messaging to a client object is supported. Using this registry, the server or another client can broadcast messages to all clients, all clients of a particular class, or all clients of a particular class and its subclasses. Finally, interest-based broadcasting allows each client to register an interest in particular nodes, or every node in the object base. Then, messages can be sent to all client objects interested in a particular node.

A variety of broadcast messages have been defined as part of the standard client protocols. These *notifications* let clients know when important events occur. For example, tools which register an interest in a file node will receive a `node-contents-changed` message when a client editor saves the contents of that file. In this case, the notification is broadcast by the client object. Other notifications,

1. Because the term "object" can refer to an object base object or a scripting language object, we generally use *node* for the former case. This is in line with standard hypermedia nomenclature, but at odds with the PCTE definition.

such as those defined for link creation and link deletion, are sent out automatically by the server to clients interested in the affected nodes.

These basic facilities for messaging, registering clients, and broadcasting form a control integration mechanism which is a superset of the broadcast message server of Field or the broadcast communication facility of Hewlett-Packard's SoftBench. The server's scripting language and the standard protocols defined in it concentrate the application-specific logic in one place, and they provide a basis for powerful customization.

The above facilities form the *foundation layer* of the server. For SoftBench-style applications, this is all that is needed, and the server can be configured to support only this layer. The *hypermedia layer* builds upon this foundation to support hypermedia applications. This layer consists of HyperLisp functions that access and manipulate the object base and allow HyperLisp messages to be sent to individual nodes and links. Node and link messages are actually interpreted by *node handler* and *link handler* HyperLisp objects. Handlers can be specified for any node or link type specified in the schema. Default handlers define link traversal, object opening, and other hypermedia operations. However, an application can define new handlers to have different forms of hypermedia. For instance, the default action on link traversal is to open a new window; this could be changed to reuse the window in which the link traversal started, or even to redefine traversal to just execute a script stored in the link instead of opening the link's destination object. Or, all traversal and open events could notify a browser that builds a map of the trails you are following through the software web. Handlers defined by an application can override some operations while inheriting others from a superclass.

In summary, the control integration facilities of HyperWeb are one of its significant contributions, giving it a high degree of flexibility and customization.

2.2 Data Integration

Our interest in PCTE centers on its Object Management System (OMS), which implements a binary Entity-Relationship-Attribute data model. Entities are called *objects*. An object has a *type*, a set of *attributes* and *links* which relate the object to other objects. Some objects have contents and are called *files*; objects are not required to have contents. Like Unix, PCTE imposes no interpretation on the contents of files, leaving this to the tools. Links are typed and have attributes; a pair of links forms a bidirectional *relationship* connecting two objects. OMS *schemas* are sets of node and link types, along with specifications of which kinds of nodes can be linked by each relationship type. The structure imposed by schemas provides us with the model for our "software webs." The OMS standard also provides transactions, locking, access control, views, transparent data distribution, and primitives for version management.

The standard schemas we have defined for HyperWeb define hyperlinks. Each end of a hyperlink can be anchored to part of a node's contents. HyperWeb stores hypermedia anchor information as attributes of the hyperlinks: it does not require the contents to be augmented with this information. When a link is created between an origin document and a destination document, the user makes selections in both documents. HyperWeb asks the client tools displaying these documents to provide strings identifying the selections. These strings are stored as attributes of the link being created. Later, when the origin document is opened, the client tool opening it is given back the selection information for all the outgoing links; the client tool uses this information to display anchors in the document.

The classical problem with large hypertexts with many interconnections is that the user can easily become "lost in hyperspace." Although this problem can be partially ameliorated with advanced browsing techniques, hypertext systems must support stronger data structuring capabilities than arbitrary graphs. While it may be generally useful to allow unrestricted linking from any node to any other node, such as software engineering, tend to have a natural structure associated with them which can be taken advantage of when designing an environment. A PCTE schema can reflect such a natural structure by limiting the types of nodes that a particular link type can join. With a carefully crafted schema, there is less likelihood of the user becoming disoriented while navigating the object base.

2.3 Presentation Integration

At the simplest level, presentation integration is provided by the OSF/Motif user interface standards. The base tools of HyperWeb (see below) all use the OSF/Motif toolkit, making them consistent in appearance and behavior.

The HyperWeb server also supports two mechanisms which help provide a unified interface for an environment. First, it can attach menus to a tool window by using the X Window System to intercept a specified keystroke and interpret that as a command to display a menu. The client object, not the client tool, controls whether this should be done. These menus can be specific to the tool and the type of the node being shown in that window. Each menu item has a HyperLisp expression associated with it which the server executes when that item is selected. This gives the user the perception that the tool has been customized for a particular environment, even though the tool itself has not been modified.

The second mechanism supporting presentation integration is a set of HyperLisp functions can be used to build and display Motif dialogues and input forms. There are more than a half a dozen dialogues including yes-no questions, confirmations, and various types of error messages. Input forms can have a wide variety of widgets added to them, including dates, sliders, push buttons, and Booleans. In practice, we have found that much of the interface for an environment can be constructed using these capabilities and provides a unified look-and-feel across the entire environment.

A major problem that integration frameworks must face is that environment-specific tools will not necessarily adhere to any particular standard. In fact, the tools being integrated may not even have the same interaction paradigm. For example, one of the problems that we have faced in supporting hypertext in an open framework is that link traversal is initiated differently from editor to editor. This has led to users occasionally being confused as to how traverse a link. One solution is for the environment builder to only integrate tools which conform to a specific user interface standard, but this would then limit the tools available in the resulting environment. This issue will be a recurring problem for environment builders.

3.0 INTEGRATED CLIENTS

The HyperWeb framework comes with several client tools already integrated. An environment developer can add to these tools as needed.

3.1 Base Tools

Currently, the base tools integrated in HyperWeb include various object base browsers and editors and two query tools. These tools were designed specifically for use with HyperWeb.

Outline web editor. The primary tool for exploring the software object base is the *outline web editor* (OWE). It gives you an outline processor view of a part of the object base centered around some "root" node. One indentation level beneath the root node are the nodes linked directly to it. Two levels of indentation indicate the nodes at a distance of two links from the root node, and so on. (Cycles in the web are detected to prevent confusing expansions.) Links beneath any node in the display can be shown or hidden. Filtering restricts the types of links shown. Navigational aids like bookmarks are also available. There may be several outline web editors open at once, each looking at the object base in a different way.

Through its menu options, the OWE provides an interface to editing operations such as adding links, creating nodes, deleting links, and copying links. Double-clicking on a node opens it and displays its contents using the client tool that manages that type of node (e.g., FrameMaker). The OWE menus can be customized: new menus can be added, and existing menus and menu items can be replaced. This enables environment-specific operations to be invoked from the OWE. Custom menus and their associated actions are specified in HyperLisp.

We chose a primarily textual representation for the object base browser to avoid the problems associated with graphical views of arbitrary hypermedia data. Graphical views can be unmanageable when hundreds of nodes are involved, and there is no single, clear mapping to two or three dimensional space as there is in, say, schematic capture. If an automatic layout algorithm is used to build the graphical view, adding a few nodes can result in large, seemingly arbitrary changes to the view, quite unlike the addition of a few index cards to an organizational layout on your desk. Further research in hypermedia ought to solve these problems, but for the time being the outline processor approach seems best for large volumes of data. One promising approach taken by a HyperWeb beta site is to develop environment-specific browsers. These use built in knowledge about the environment's data model to present coherent graphical views.

OMS tools. These are dialogues for creating new nodes, editing a node's type and attributes, creating and editing links, and choosing a link from a list of links leaving a node. They provide a user-friendly interface to the full features of the PCTE OMS. In mature applications, specialized menus and scripts tend to be used instead.

Query tools. Hypermedia browsing is a navigational access of data. To find a node, you must know a path to it or guess where it is. The outline web editor makes this easier, but you still have to search for nodes. Therefore HyperWeb also supports browsing via a navigational query language for PCTE. The language supports queries based on attribute values of nodes and links, node and link types, node contents searches, and structural cues. The query interpreter has a HyperLisp interface, allowing queries to be used in scripts. Environment developers attach these scripts to menu selections to provide "canned queries."

HyperWeb comes with two user interfaces for building and executing queries. There is a straight-forward query interface in which you type a query, and get a window showing the result. To open one of the result nodes, you click on it. In most cases users will have simple queries and will not want to learn a query language. For these users, we have a simple point-and-click interface that makes it easy to enter simple queries. This interface presents a Query-By-Example style interface where you specify the starting node(s) and the kinds of links you want to follow out from them. You then indicate the attributes and nodes you want printed out by clicking on their symbols. For instance, you can easily ask for all level "A" modification requests (MRs) related to any subsystem in a given software project. The result will be a list of the MRs, and you can look at each one by clicking on it in the result window.

3.2 External Tools

One of the goals of HyperWeb was to allow existing development tools to work within the framework. So rather than create a new set of editors, we have integrated two existing editors. Epoch is the primary editor for text nodes [Kapl90]. Epoch is based on the popular GNU Emacs text editor, and adds support for multiple X windows, mouse operations and attributed regions called *buttons*. The second editor, FrameMaker, provides document-editing capabilities, including structured graphics.

For both editors, the main task of integration was to implement the client object methods specified by HyperWeb's standard protocols. Typically, a client object method consists of a few lines of Hyperlisp and usually contains a call or event message to the client process. For Epoch, the body of these messages consist of expressions in Epoch's own customization language, *elisp*. For example, to implement the `select-next-text-pattern` method, Epoch's client object sends an event message to it consisting of the elisp expression `(search-forward` *`path pattern`*`)`. In some cases, the elisp expression is a standard Epoch function, but in others it is a function written for the HyperWeb integration.

As we integrated these editors, we found several desirable characteristics that an editor should have to be integrated into any hypermedia framework. First, the tool must be able to "communicate" with other programs. If it cannot, it can be invoked, but little else. The *vi* editor is a good example of this. Secondly, the tool should have an interface element that can be used as a hypermedia anchor. Epoch provides buttons, which meet this criteria. Framemaker has hypertext markers. A tool lacking this characteristic can still be useful when integrated as a non-hypermedia client that responds to control messages. Finally, if the tool is extensible (typically via a scripting language), it is much easier to integrate. In this regard, Epoch has much stronger support than Framemaker, allowing us to implement the entire HyperWeb node editing protocol. Using these criteria, tools such as the Interleaf document editor and IDE's Software Through Pictures™ design tool are potential hypermedia clients.

3.3 Environment-Specific Tools

An environment can integrate new general- or special-purpose client tools. For example, a group building a Cobol development environment designed a client that manages connections to remote, non-Unix machines and transfers files between those machines. In most cases, the environment-specific clients are implemented as external programs which communicate with the server through a client object. However, in some cases, environment-specific clients can be constructed entirely in HyperLisp. These are usually "agents" which perform actions based on notifications.

4.0 THE UNIX DEVELOPMENT ENVIRONMENT

Our Unix development environment ("UDev") extends the HyperWeb framework to support general Unix development and maintenance. Rather than duplicating Unix functionality in an isolated environment, we augment it. The environment uses document linking and hypermedia to make systems more understandable and to preserve the developer's knowledge of the system. Because the environment has been built using HyperWeb's scripting language, (except for environment-specific tools), we have been able to quickly add new functionality and try out new ideas.

The basic "UDev" development process is illustrated in Figure 2. In the first step, the user imports existing development directories into the object base. The import operation uses a pattern matching algorithm to map each Unix file to an appropriate node and link type in the object base. Next, during normal editing, the user "elaborates" the web, decomposing files and adding annotations. When the user decides to compile, the web must be "tangled" to create compilable units and then exported to the Unix filesystem where it can be operated on by the standard Unix tools. Finally, since the user is familiar with the elaborated web and not the tangled version, it is necessary to map compiler error messages and debugger messages back into the web. This "tangle traceability" maps locations in the tangled and exported files to familiar locations in the nodes.

4.1 Elaborating the Web

After importing an existing set of files, the OWE is used to browse through the project. A source file or document can be opened for editing by double-clicking on its OWE entry, bringing up Epoch or FrameMaker. As part of the editing process, the web is "elaborated." This refers to the process of restructuring and documenting the code using hypertext links. The goal of this activity is to create a "web" of links that makes the interconnections between documents comprising the system explicit.

In order to simplify link creation, we wrote scripts that streamlined it for three operations: *annotation, decomposition*, and *refinement*. These operations are invoked from menus attached to node editor windows and operate on the current selection. The scripts that implement the operations are not specific to a particular node editor. They use only

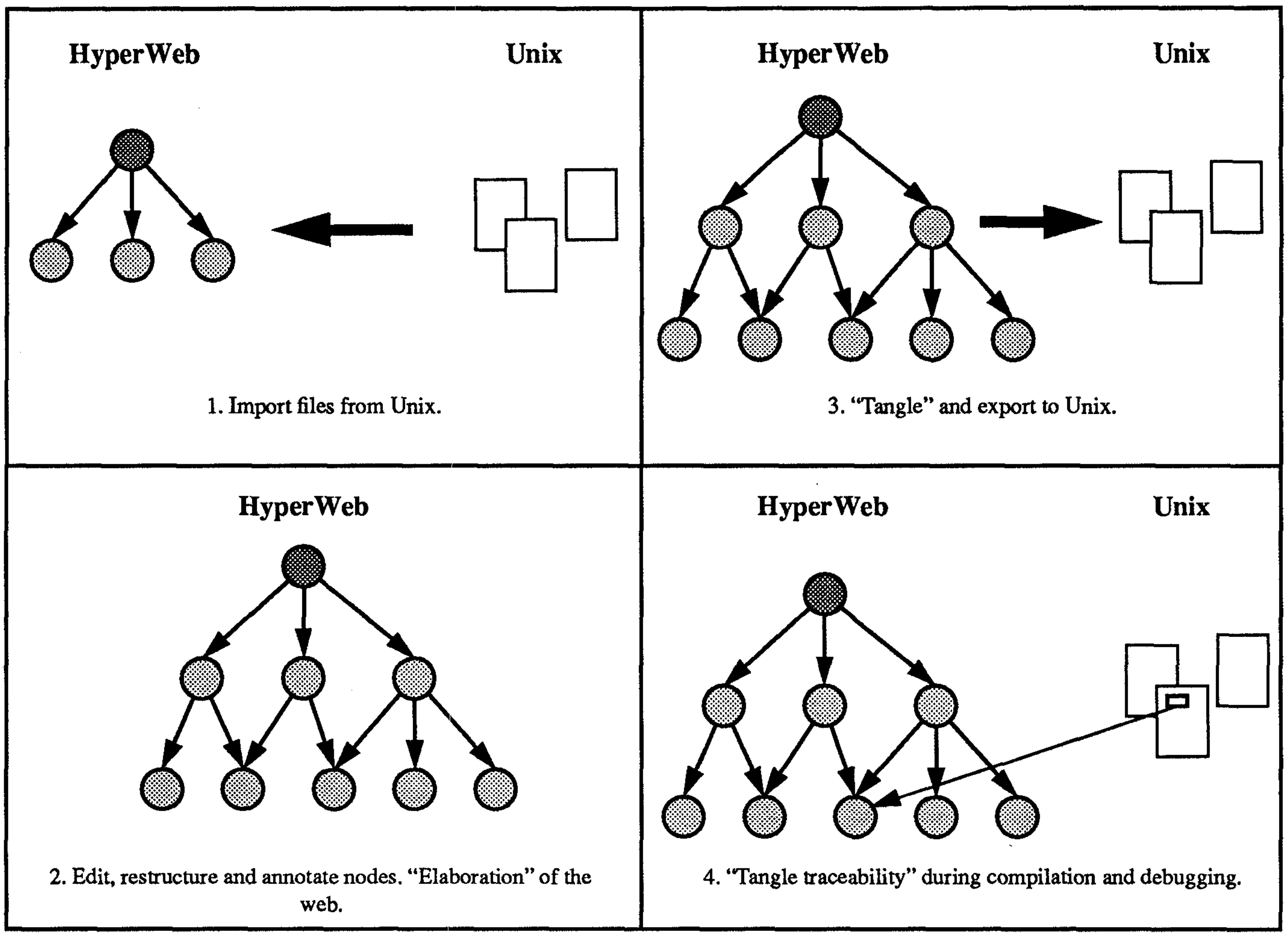

HyperWeb
Unix
1. Import files from Unix.
HyperWeb
Unix
3. "Tangle" and export to Unix.
HyperWeb
2. Edit, restructure and annotate nodes. "Elaboration" of the web.
HyperWeb
Unix
4. "Tangle traceability" during compilation and debugging.

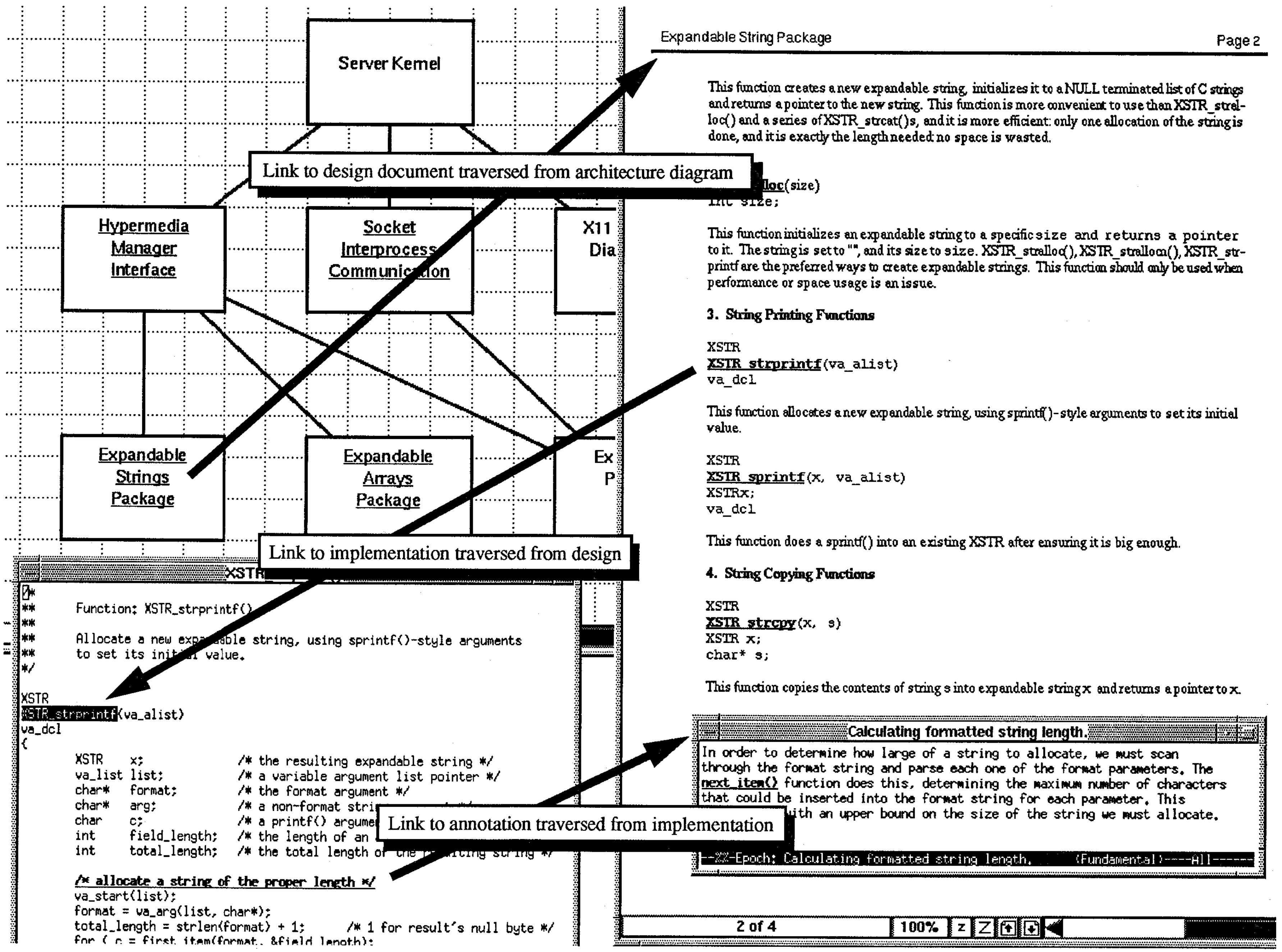
FIGURE 2. Udev Development Process

methods defined as part of HyperWeb's standard client protocols for text editing. They would work with any editor implementing the full text-editing protocol.

Annotation. Annotations can be used to attach many different kinds of comments to a source file in a manner analogous to placing a yellow "sticky note" on a source listing. A comment might address the style of the code, or it might discuss alternative design approaches. Annotations can be used to attach reminders about enhancement ideas and needed changes. Multiple annotations can be attached to the same anchor. Activating an anchor with multiple anchors will bring up a link selection dialogue, allowing you to select and open the annotation of your choice.

Decomposition. Decomposition is a technique that can be used to break large files into more manageable (and meaningful) sections. It allows you to isolate conceptual units of code and exclude irrelevant details, making the structure and meaning of the code clearer. When you decompose a section of code, it is replaced by a short description of the code that serves as the anchor for a hypertext link. The code is placed in a new node which can be reached by traversing the link. For example, you might want to decompose each function in a source file into a separate node. This would allow you to use the top level node like a table of contents, where any function can be displayed with a single click. Another strategy would be to use decomposition to isolate groups of related functions, such as "Private" and "Public" functions of a library. Or you could use decompositions to hide complex code that makes a program difficult to read. The *recompose* operation reverses decomposition, bringing the code back into the parent node.

Sharing decompositions provides the basis for a first level of reuse. Instead of creating a new decomposition, you might identify a place where an existing decomposition would do the job, and simply create a link to it. The next level of reuse would be the frame-based approach along the lines of [Bass87]. Currently, when a program is decomposed into sub-parts, those sub-parts are included literally when the program is recomposed for the compiler. In frame-based reuse, a sub-part is a reusable code *pattern*, and a decomposition link from a program to that pattern would carry parameters and directives for tailoring it. Changes to the pattern are automatically propagated to the program each time it is recomposed.


Server Kernel
Hypermedia Manager Interface
Socket Interprocess Communication
X11 Dia
Expandable Strings Package
Expandable Arrays Package
Ex P
Link to design document traversed from architecture diagram
Link to implementation traversed from design
Link to annotation traversed from implementation
XSTR
2 of 4
100%

```
/*
**      Function: XSTR_strprintf()
**
**      Allocate a new expandable string, using sprintf()-style arguments
**      to set its initial value.
*/

XSTR
XSTR_strprintf(va_alist)
va_dcl
{
        XSTR    x;              /* the resulting expandable string */
        va_list list;           /* a variable argument list pointer */
        char*   format;         /* the format argument */
        char*   arg;            /* a non-format stri
        char    c;              /* a printf() argume
        int     field_length;   /* the length of an
        int     total_length;   /* the total length of the resulting string */

        /* allocate a string of the proper length */
        va_start(list);
        format = va_arg(list, char*);
        total_length = strlen(format) + 1;      /* 1 for result's null byte */
        for ( c = first_item(format, &field_length);
```

Expandable String Package Page 2

This function creates a new expandable string, initializes it to a NULL terminated list of C strings and returns a pointer to the new string. This function is more convenient to use than XSTR_strelloc() and a series of XSTR_strcat()s, and it is more efficient: only one allocation of the string is done, and it is exactly the length needed: no space is wasted.

```
...lloc(size)
int size;
```

This function initializes an expandable string to a specific `size` and returns a pointer to it. The string is set to "", and its size to `size`. XSTR_strallo(), XSTR_strallocn(), XSTR_strprintf are the preferred ways to create expandable strings. This function should only be used when performance or space usage is an issue.

3. String Printing Functions

```
XSTR
XSTR_strprintf(va_alist)
va_dcl
```

This function allocates a new expandable string, using sprintf()-style arguments to set its initial value.

```
XSTR
XSTR_sprintf(x, va_alist)
XSTRx;
va_dcl
```

This function does a sprintf() into an existing XSTR after ensuring it is big enough.

4. String Copying Functions

```
XSTR
XSTR_strcpy(x, s)
XSTR x;
char* s;
```

This function copies the contents of string s into expandable string x and returns a pointer to x.

Calculating formatted string length.

In order to determine how large of a string to allocate, we must scan through the format string and parse each one of the format parameters. The next_item() function does this, determining the maximum number of characters that could be inserted into the format string for each parameter. This ...ith an upper bound on the size of the string we must allocate.

```
--%%-Epoch: Calculating formatted string length.     (Fundamental)----All------
```

FIGURE 3. Documentation, source and annotations linked using HyperWeb.

Refinement. Refinement allows you to type in a pseudocode description of a portion of code and later use that description as the anchor of a decomposition link. Using this approach, the structure of your finished product can better reflect the thought process used in creating it. If, after "fleshing out" the actual code in a refinement, you decide that it should really be part of the parent node, it can be recomposed.

Figure 3 depicts an example session with HyperWeb illustrating linking between various nodes. In this picture, the user has opened a nôde containing an architecture diagram. The module labelled "Expandable Strings Package" is an endpoint of a link to a design document, which the user has traversed. This document contains several function specifications. Then the user traversed an `implements` link from the specification document to a node containing source code, and finally traversed an `annotation` link to view a comment about the implementation.

4.2 Exporting and Tangle Traceability

When the developer has made some changes and is ready to compile and test the program, the changed nodes are extracted and exported back to the Unix file system. Here, the developer can compile and debug the program in the usual fashion. The source files are *not* re-imported after building and testing; the developer simply returns to the web for further editing.

If source files have been decomposed during editing, they must be "tangled" into sequential form before being compiled. A utility that performs this function is automatically invoked when a node is exported to the Unix file system. When working with exported files, the "Tracer" client can map line numbers from a compiler error message or debugger breakpoint message to the specific node in the object base which contains that line. "Tracer" works by parsing the error message to obtain the line number and file name. It then consults a mapping file generated during the "tangle" operation to determine which node contains the correct line and then asks the node to open itself at that line. If the debugger is integrated with HyperWeb, the programmer does not even need to initiate the mapping operation; instead, the debugger can send the appropriate message to the Tracer client.

4.3 Other Elaborations of the Web

We have also been exploring other ideas for "elaborating" the web. For example, by maintaining modification requests (MRs) in the object base with other software artifacts, changes can be linked to the MRs that initiated them. A person browsing the object base can easily determine which problems relate to which changes because those relationships are explicit. As a developer works on the modification to resolve an MR, he can annotate it with excerpts from debugging sessions or examples of code causing the problem. The MR system is tied to the development process using scripts that are triggered by notifications broadcast when a version of a component is checked in or out. These scripts give developers the option to open or close an MR at that time.

We are already using the annotation capability described above to do on-line review of design documents in order to capture the rationale behind our design decisions. A full-fledged on-line inspection application would be a straightforward extension of this capability. The combination of HyperWeb's hypermedia capability and the PCTE OMS also lends itself to tracing requirements from initial requirements documents to design documents and finally to source code.

5.0 CONCLUSIONS

Our experience with the HyperWeb framework has been encouraging. The combination of a hypermedia integration framework, a flexible scripting language with interface support, and the PCTE OMS has allowed us to construct an environment that supports our idea of software as a rich web of interconnected components. This environment moves beyond treating software as a collection of isolated objects by enabling developers to store a variety of software artifacts and their relationships in a software object base. It supports operations that improve the readability of code. It is flexible enough to support different ways of developing and maintaining software.

The HyperWeb framework was released as a product called the PCTE Workbench™ in July 1992. The UDev environment is currently in the beta testing stage. We plan to continue improving the environment and to integrate additional clients including a design tool and a code analysis/debugging tool.

ACKNOWLEDGMENTS

This work was supported in part by Groupe Bull and the U.S. Department of Defense, Fort Meade, Maryland. We also thank Dan Campbell, Al DuPuis, James Kiso, François Leygues, Cameron McLean, Gérard Memmi, Régis Minot, Rick Young, and the many others who have made valuable contributions in the design, implementation, and beta testing of HyperWeb.

References

[Bass87] Bassett, P.G. Frame-based software engineering. *IEEE Software, 4,* 4 (July 1987), 9-16.

[Bigl88] Bigelow, James. Hypertext and CASE. *IEEE Software*, March 1988, 23-27.

[Boud88] Boudier, G., Gallo, F., Minot, R. and Thomas, I. An overview of PCTE and PCTE+. *ACM SIGSOFT Software Engineering Notes, 13,* 5 (February 1989), 248-257.

[Conk87] Conklin, J. Hypertext: An introduction and survey. *IEEE Software*, (September 1987), 17-41.

[Cybu92] Cybulski, J. L., and Reed, K.A. Hypertext based software-engineering environment. *IEEE Software, 9,* 2 (March 1992), 62-68.

[Deli86] Delisle, N. and Schwartz, M. Neptune: a hypertext system for CAD applications. In *Proceedings of the ACM SIGMOD* '86 (Washington, D.C., May 28-30, 1986), 132-142.

[ECMA90] *Portable Common Tools Environment (PCTE), abstract specification, ECMA Standard 149.* European Computer Manufacturers Association, Geneva, December 1990.

[Ferr89] Ferrans, J.C. and Memmi, G. HyperWeb: conceptual overview and system architecture. Vista Technologies and Bull HN Technical Report, September,1989.

[Hala87] Halasz, F.G., Moran, T.P., and Trigg, R.H. Notecards in a nutshell. In *Proceedings of the 1987 ACM Conference on Human Factors in Computer Systems* (Toronto, Ontario, 5-9 April), 1987, 45-52.

[HP89] *Exploring HP SoftBench: a beginner's guide.* Part B1622-90001. Hewlett-Packard, Ft. Collins, Colorado, 1989.

[Kapl90] Kaplan, S. Epoch: GNU Emacs for the X windowing system, release 3.2. Technical Report, Department of Computer Science, University of Illinois at Urbana, 1990.

[Knut84] Knuth, D. Literate programming. *Computer Journal, 27,* 2 (May 1984), 97-111.

[PCTE88] *PCTE, a basis for a Portable Common Tool Environment, functional specification, version 1.5.* PCTE Interface Management Board, Brussels, 1988.

[Reis90] Reiss, S.P. Connecting tools using message passing in the Field environment. *IEEE Software*, 7, 4 (July 1990), 57-66.

[Ritt73] Rittel, H., and Webber, M. Dilemmas in a general theory of planning. *Policy Sciences,* Vol. 4, 1973.

[Thom92] Thomas, I. and Nejmeh, B.A. Definitions of tool integration for environments. *IEEE Software, 9,* 2 (March 1992), 29-35.

[Uttn89] Utting, K, and Yankelovich, N., Contexts and orientation in hypermedia networks. *ACM Transactions on Informations Systems*, 7, 1 (January 1989), 58-84.

[Dam88] Van Dam, A., Hypertext '87 keynote address.. *Communications of the ACM*, 31, 7 (July 1988), 887-895.