A Practical Approach towards Active Hyperlinked Documents

Eckhart Köppen, Gustaf Neumann
Information Systems and Software Techniques
University of Essen, Germany
`Eckhart.Koeppen@uni-essen.de, neumann@wi-inf.uni-essen.de`

This paper presents an implementation for web-based, active documents. Although several approaches to distributed, active documents exist already, we decided to establish a new model which provides more flexibility and interoperability without giving up formality. The model is based mainly on the Extensible Markup Language and makes use of the Document Object Model, Cascading Style Sheets and the Intrinsic Event Model, which are all open standards defined by the W3 Consortium.

1 Motivation

The rapid success of the World Wide Web has led to a new class of applications which are constructed using `HTML` [20] for the user interface and `CGI` scripts for the application's logic. They have a more or less strong resemblance to mainframe programs: the user enters data into a form which is transferred to the web-server, evaluated and the results are passed back to the user agent. As a result, the computational load and the application logic are located entirely on the server side.

In contrast to server-centered web applications, a client-centered application model has emerged through the use of scripting languages such as JavaScript [16]. Interfaces to the user agent and the current document exist in the form of plug-ins [15], Java applets [4] and embedded scripts [12]. However, with most of these solutions a number of problems exist: plug-ins are strongly tied to the chosen user agent and the client platform, the interface to the document is in all cases either non-existant or allows only the changing of text and there are still distinctions between client- and server-side application logic.

To gain more flexibility and to remove any distinctions between server and client, the approach which we will present in this paper incorporates the application code into the `HTML` document, thus turning the formerly passive document into an application itself. We will refer to these enabled `HTML` documents as *active, hyperlinked documents* (`AHD`s). With this architecure, a different application model can be implemented. This starts with small programs like a bookmark page which controls its logic and appearance itself and goes up to workflow management systems which contain mostly independant documents with different states and possible operations on them. More generally speaking, the possible uses of `AHD`s ranges from controlling the contents and layout of a single document to support of coordination and collaboration techniques.

2 System Overview

The general functionality which is needed by this document-centered architecture covers several aspects: an interface to the user agent, flexible structuring and semantic annotation of the document (state, behavior, presentation), introspection through an interface to the document itself, a layout and a scripting mechanism. Figure 1 shows these building blocks.

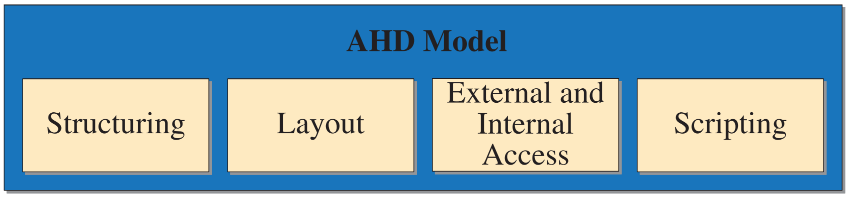

Figure 1: Building Blocks of an AHD model

We will be using two standard proposals which have been introduced recently by the W3 Consortium: the *Extensible Markup Language* (XML) and the *Document Object Model* (DOM) (see [2] and [3]) . The DOM provides a clear, programming language independant interface to the document structure. Additionally, it defines interaction with the user and the user agent through the *Intrinsic Event Model* (IEM). As a means to annotate a document semantically, HTML is too limited. Here, XML will be used as the basis for the semantic annotations and as a structuring language.

The techniques used in the basic building blocks of the AHD model are shown in Figure 2, note however that the use of Tcl and OTcl ([19], [21]) as the scripting language is not a mandatory characteristic of an AHD. Tcl/OTcl is chosen here because it has a reasonable balance of simplicity and capability.

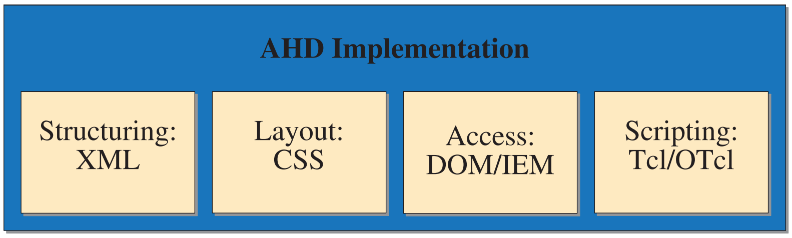

Figure 2: Characteristics of an AHD

The proposed model for AHDs is embedded in a system architecture which is a natural extension to the architecture of current web-based applications. In those systems, web servers provide access to a repository of either statically available or dynamically created documents. Web browsers are used as user agents to request the documents from the web servers.

On the client side, the user agent will be extented to incorporate a *runtime environment* (RE). The RE provides an interface from the user agent to the AHD and vice versa. In addition, it provides the functionality for introspection to the AHDs.

On the server side, the RE will be incorporated into the web server (this step will not be discussed in this paper), so that certain functions in an AHD can be executed prior to its transfer to the user agent. An AHD could also reside in the web server for a longer time, responding to requests itself.

The basic mechanisms needed to implement our model of AHDs will now be described.

2.1 XML

The Extensible Markup Language is very closely related to the Hypertext Markup Language and is a subset of the *Standard Generalized Markup Language* (SGML, see [5]). Some of the main goals for the design of

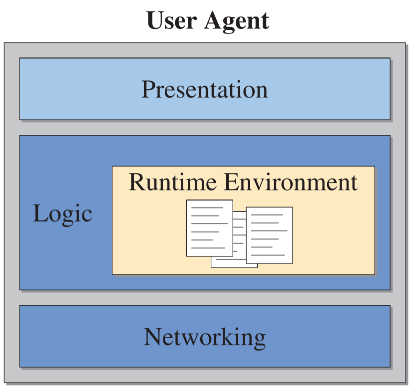

Figure 3: User Agent Architecture

XML were:

- straightforwardly usable over the Internet
- upwards compatibile with SGML
- easy implementation of XML processors

Like SGML, XML documents are composed of physical units (entities) and have a logical structure which is formed by elements. Elements are declared in a Document Type Description (DTD) and marked with start- and end-tags in the document.

For our approach, the main advantage of XML over HTML is the ability to declare elements which are needed to form an active document. In comparison to SGML, XML is explicitly targeted at the World Wide Web and more widespread support from content creators and software developers is expected.

2.2 DOM

The structure of HTML and XML documents can be accessed through the Document Object Model (DOM). The DOM describes the parts of a document in terms of nodes which are organised in a tree. Here, the more interesting node types are *text*, *elements* and *attributes*.

The interface to the different nodes is described via the *Interface Definition Language* (IDL) from the Object Management Group [18]. An interface definition contains the attributes and possible operations for the node types. Corresponding definitions can be derived for various languages, e.g. Java. Among the operations defined for the node types are operations to create new elements, manipulate their list of children and modify their attributes.

2.3 Intrinsic Event Model

To map events to behavior, we use the *Intrinsic Event Model* (IEM) defined in the DOM. Events are tied to the element where they occur. If an element does not process an event, it is propagated to its parent element.

Any activity is triggered by external events. They can be grouped into user events and events caused by the runtime environment. User events are pointer events (motion, clicks), keyboard events (key pressed, key

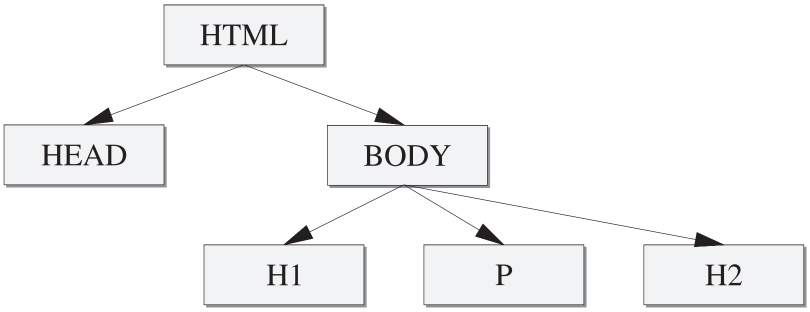

Figure 4: Typical node tree (elements only)

released), form related event (list selections, text changes), and focus changes. Events triggered by the RE are document loading and document unloading.

2.4 Style Sheets

Since XML defines only the structure of a document, a separate layout definition is provided through style sheets. The W3 Consortium proposes *Cascading Style Sheets* (CSS) to be used together with HTML and XML [13].

Style sheets imply a new set of attributes which are associated with an element. These attributes control the visible aspects of the elements, e.g. margins, borders, text styles. In addition, a style sheet can also describe aural properties for elements to enhance accessability of a document.

3 Architecture

To implement the infrastructure for AHDs, basic definitions are needed to describe the structure and semantics of an AHD. In addition, a mapping of the event model to the document behavior has to be defined.

3.1 Document Structure

The structure of an AHD is defined in a DTD. To allow the RE agent to process an AHD, two approaches are possible. On the one hand, the DTD itself could declare a set of elements which are recognized by the user agent and contain any code and data required for the AHD. In our model, we chose a more flexible approach. Here, any element can contain code and data. For this purpose, the DTD declares only two specific elements, namely <FUNC> and <VAR>. They will be explained in detail in the next section. Instances of these elements are always associated with their parent element, i.e. their scope is the parent element. This association is established by the RE, which manages the access to code and data for each element.

The following sample illustrates the mechanism. The first part of the example declares a DTD with the elements <FUNC> and <VAR>:

```
<!ELEMENT ahd o o ANY>

<!ELEMENT func - - CDATA>
```

```
<!ATTLIST func
            name     CDATA #REQUIRED
            type     CDATA #IMPLIED>

<!ELEMENT var    - - CDATA>
<!ATTLIST var
            name     CDATA #REQUIRED>
```

In the second part, two additional elements (<order> and <buyer>) are declared. An instance of the <order> element is created. It contains the function `print_header` and instance of the element <buyer> with two functions (`print` and `duplicate`) and two variables (`name` and `email`):

```
<!DOCTYPE ahd [
<!ELEMENT order - - (buyer | FUNC | VAR)*>

<!ELEMENT buyer - - (#PCDATA | FUNC | VAR)*>
<!ATTLIST buyer
            id       CDATA #IMPLIED>
]>

<order id="o80">
        <func name="print_header"> ... </func>
        <buyer id="p80">
                <func name="print"> ... </func>
                <func name="duplicate"> ... </func>
                <var name="name">John Doe</var>
                <var name="email">doe@uni-essen.de</var>
        </buyer>
</order>
```

The `RE` associates both the functions and the variables with the <buyer> element. As a consequence, the variables and the functions can be accessed only through the <buyer> element. The <buyer> element itself can be addressed using its `id` attribute. The addressing scheme for elements is defined in more detail in the `XML` specification [2].

The local scoping of functions and variables requires a method to faciliate access to functions and variables. We will use *parent delegation* to look up function and variable elements. In order to access an element through its `name` or `id`, the parent chain of the current element will be searched. Function and variable elements are associated with an element if they are a direct child of this element. In the example, the variable `name` can be modified from within the function `print` since it is a direct child of the <buyer> element.

Figure 5 shows the lookup of the function `print_header` which is called in the function `print` of the element <buyer>. Here, the lookup of the function is continued in the parent of <buyer>, <order>. This element contains the wanted schript as a direct child.

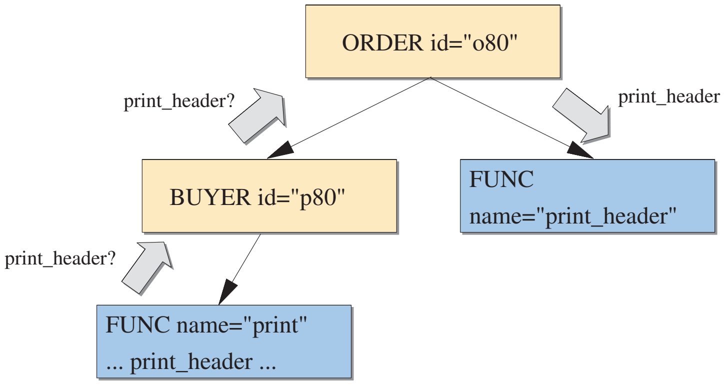

Figure 5: Parent delegation

3.2 Structural Elements

The two elements which are used to incorporate state and behavior into an `AHD` are `<FUNC>` and `<VAR>`. They contain either function code or variable values and are a logical part of their parent element. It is important that any other element which will contain these elements is defined appropriately by including the `<FUNC>` and `<VAR>` elements in the content declaration. Both elements have the `CSS` display property set to `none` so that they will not be layouted and displayed.

3.2.1 FUNC Element

The declaration of the `<FUNC>` element is as follows:

```
<!ELEMENT FUNC - - CDATA>
<!ATTLIST FUNC
              name        CDATA       #REQUIRED
              type        CDATA       #IMPLIED>
```

The `name` attribute contains the name of the function under which it can be called, and the `type` attribute describes the *Internet Content Type* of the function (which is basically the programming language used). The contents of the `<FUNC>` element (i.e. the text between the start- and end-tag) is the code of the function. The name of a function has to be locally unique, which means that the parent element does not have any other function with the same name. If another function with the same name exists, only the first element is considered.

3.2.2 VAR Element

The following code shows the declaration of the `<VAR>` element:

```
<!ELEMENT VAR - - CDATA>
<!ATTLIST VAR
              name        CDATA       #REQUIRED>
```

The element has only one attribute, `name`. Like the <FUNC> tag, this contains a locally unique name for the variable. The contents of the <VAR> element is the value of the variable.

3.3 Event Model

The events defined in the event model are mapped implicitly to element functions. Any function which name equals an event name will be called when that event occurs. According to the IEM an event handling function can also be defined in an attribute which has the name of the handled event. The next examples (which are all equivalent) show the usage of event handlers.

```
<label1>
            <func name="onclick"> echo Mouse clicked! </func>
            ...
</label1>

<label2 onclick="echo Mouse clicked!">
            ...
</label2>

<label3 onclick="mouse_handler">
            <func name="mouse_handler"> echo Mouse clicked! </func>
            ...
</label3>
```

Each event handler is passed additional information about the event, e.g. the pressed key or the pointer location.

3.4 Programming Interface

An element consists of five components as shown in Figure 6.

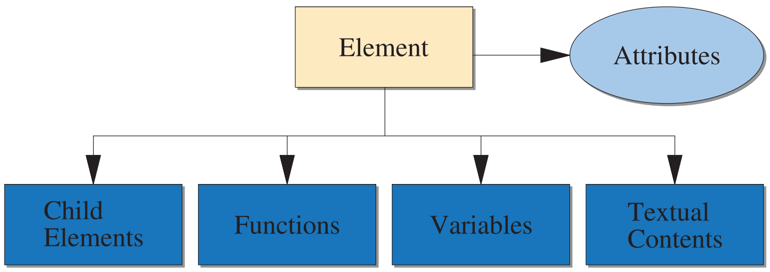

Figure 6: AHD Element components

The access to the attributes and the child elements from within the RE and and AHD is provided through the functions defined in the DOM. These functions can also be used to access the functions, variables and the textual contents of an element, but to ensure the *parent delegation* mechanism for function and variable access, convenience functions are implemented as well. These are:

setVar: sets a variable value (the textual contents of a <VAR> tag)

getVar: gets a variable value (the textual contents of a <VAR> tag)

setContents: sets the textual contents of an element. If the element contains other child elements, they will be deleted (to achieve more control over the the contents of an element, using the functions of the DOM is recommended).

getContents: gets the textual contents of an element. Note that any child elements in the content are ignored (e.g. the contents of the <p> tag in "<p>A nested tag</p>" is "A tag").

Since the execution of an element function can not be triggered by DOM functions, the RE exports another utility function:

callFunc: calls an element function

The function interfaces are defined independantly from any programming languages. However, in our implementation, they will be written in C.

4 Implementation

The implementation of the model is mainly tied to the implementation of the user agent. We developed the extensible web browser Cineast [11] to develop and evaluate novel approaches like AHDs. The main part of the Cineast is written in OTcl, this is also why we chose Tcl/OTcl as the scripting language for AHDs. The Cineast is currently running under Unix, but its main parts can be ported to other operating systems and the model for the AHDs is platform independant.

The basis of the Cineast is the prototyping environment Wafe [17]. It combines Tcl as a scripting language with different widget sets such as the Athena widget set or the Motif widget set. In addition, other libraries are linked into Wafe, among these are SSLeay [9], LDAP [7] and OTcl. For the implementation of the Cineast, a special purpose widget called Kino [10] is integrated into Wafe to handle the parsing, layout and display of XML source text. It is implemented in C to achieve acceptable performance. The roles of Wafe and the Kino widget in the different layers of the user agent are discussed below. Figure 7 shows an overview.

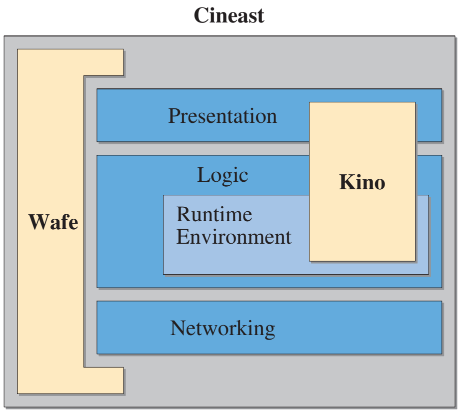

Figure 7: Wafe and the Kino widget in the Cineast

4.1 Runtime Environment

Most of the functionality required in the RE is provided by the Kino widget. The main task of the Kino widget in the RE is to parse any source text and to maintain an internal representation of the element tree. It also implements all of the DOM functions and the additional functions needed to access the functions and variables of the elements.

The Kino widget is made up of three components: the *Parser*, the *Layouter* and the *Painter*. In the RE, only the *Parser* is of interest. It produces a tree of *parsed data* (PData) elements. The Kino widget uses four different types of PData elements: a *generic element*, a *box element* which can contain children, a *text element* and an *inset element* which can be used to insert other widgets and images into the element tree.

The most interesting element is the box element. Besides its role as a structuring element to contain other elements, it holds the CSS attributes. The PData box elements correspond directly to the XML elements in the document, i.e. any XML element results in a PData box element. Figure 8 shows a part of a PData tree.

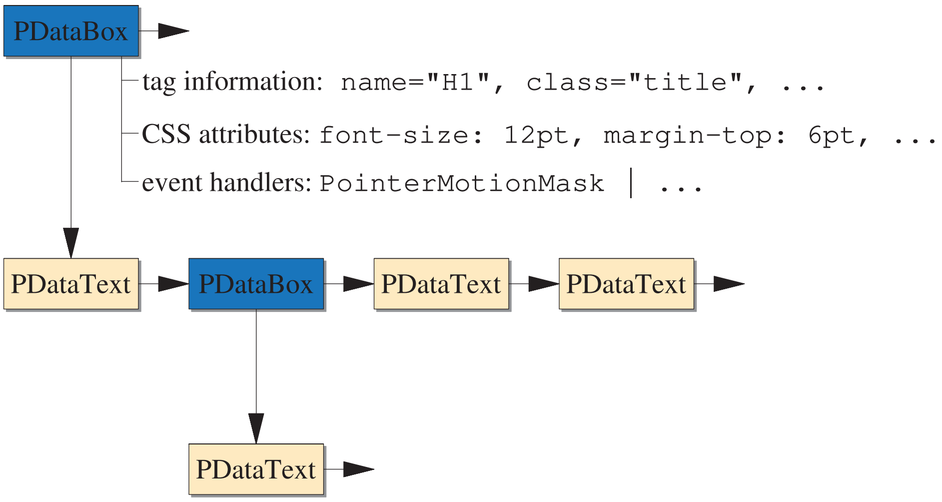

Figure 8: PData tree

Navigation in the PData tree is possible through the DOM on the one hand, on the other hand, the components of the PData elements can be used directly through their C pointers.

The Kino widget is extensible in two ways: the application can register a *tag callback*, which is called whenever a tag is encountered during the parsing process. In this callback, the application can for example modify the PData tree. The second callback which is used handles the execution of scripts. The Kino widget calls this *script callback* whenever a script has to be executed, e.g. when an event occurs or a script is called via callFunc. This makes the Kino widget independent from the chosen scripting language.

4.2 Presentation

The presentation layer of the Cineast is implemented in the *Layouter* and *Painter* components of the Kino widget. They are not needed for the basic functionality of an AHD and can be omitted if the RE is to be incorporated into a web server.

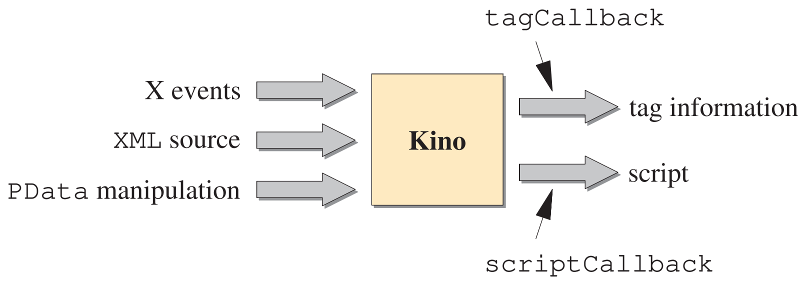

Figure 9: Interacting with the Kino widget

The Layouter works together with a CSS database. The CSS database is built during the parsing process and contains all style information. Currently, the CSS database is implemented completely in OTcl. The Layouter positions any element so that no calculations are needed for display by the Painter.

The Kino widget and the Cineast handle most of HTML 3.2, including important features like tables, images and forms. The internal layout model is box-oriented, so that a PData box results in either a block-level or inline element according to the CSS specification. In contrast to a simple flow-oriented model, boxes can be nested. Figure 10 shows the most important arrangements.

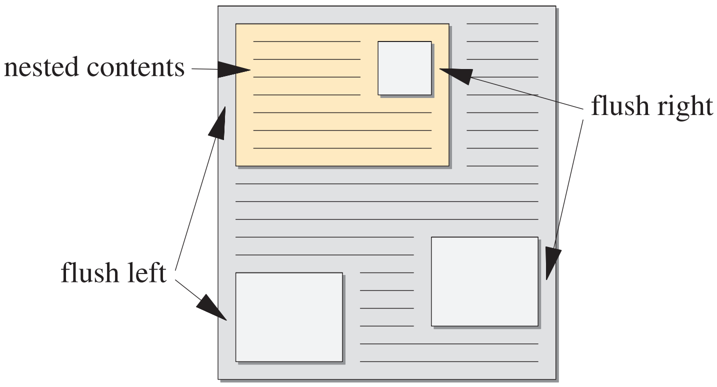

Figure 10: Some possible layout arrangements

The box model allows the realization of more complex layouts such as tables. Due to the increased complexity, incremental layout is not supported, but may be available in the future, as well as absolute positioning of boxes *within* a text flow and not on separate layers.

4.3 Application Logic

...

4.4 Networking

...

5 Conclusion

Ideas for applications which are based on web techniques such as HTML and CGI were presented early in the history of the web. In [6], the authors already point out the need for more control on both the client and server side. Related research can also be found in [8], where the Mosaic web browser is extended to be able to execute Tcl scripts. However, a more formal model is not presented.

Our proposed model for *Active Hyperlinked Documents* and the implementation of the overall system differ in two key points from existing approaches:

- since all used mechansisms and techniques are either open standards or freely available and usable software, the concept of an AHD is independant of commercial influence and easy to use and validate for others
- the system itself is constructed in a way that makes modifications very simple while still supporting a certain level of formality

More important however are the next steps that have to be made to prove the potential of AHDs. First, a formal model for the use of AHDs in distributed applications has to be developed. Starting points can be frameworks like [14]. A second step is the reduction of the distinctions between user agents and web servers by incorporating the RE into the web server. Lasty, security issues have to be addressed by including standard security techniques like electronic signatures, encryption and access models.

References

[1] T. Berners-Lee, R. Fielding, H. Frystyck: *Hypertext Transfer Protocol - HTTP/1.0* Informational RFC, <http://www.w3.org/Protocols/rfc1945/rfc1945>, May 1996.

[2] Tim Bray, Jean Paoli, C.M. Sperberg-Queen: *Extensible Markup Language (XML)*, W3C Working Draft, <http://www.w3.org/TR/WD-xml>, November 1997

[3] Steve Byrne: *Document Object Model (Core) Level 1*, W3C Working Draft, <http://www.w3.org/TR/WD-DOM/level-one-core-971009>, October 1997

[4] James Gosling, Henry McGilton: *The Java Language Environment*, <http://java.sun.com/docs/white/langenv/>, May 1996

[5] Charles F. Goldfarb. *The SGML Handbook*, Oxford University Press, Oxford 1990.

[6] Henry Houh, Cris Lindblad and David Wetherall: *Active Pages: Intelligent Nodes on the World Wide Web*, Proceedings of the First World Wide Web Conference, Geneve, 1994

[7] T. Howes and M. Smith: *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*, Macmillan Technical Publishing, 1997.

[8] M. Frans Kaashoek, Tom Pinckney and Joshua A. Tauber: *Dynamic Documents: Extensibility and Adaptability in the WWW*, Proceedings of the Second International World Wide Web Conference, Chicago, 1994.

[9] T. J. Hudson and E. A. Young: *SSLeay and SSLapps FAQ (Draft)*, `<http://www.psy.uq.edu.au/ ftp/Crypto/>`, September 1997.

[10] Eckhart Koeppen: *Entwicklung eines erweiterbaren Widgets zur Anzeige von HTML-Texten*, Master's Thesis, University of Essen, Germany, 1996.

[11] Eckhart Koeppen, Gustaf Neumann, Stefan Nusser: *Cineast - An extensible Web Browser*, Proceedings of WebNet 97, Toronto 1997

[12] Jacob Levy: *A Tk Netscape Plugin*, Proceedings of the Fourth Annual USENIX Tcl/Tk Workshop, Monterey 1996

[13] Håkon Wium Lie and Bert Bos: *Cascading Style Sheets, level 1*, W3C Recommendation, `<http://www.w3.org/TR/REC-CSS1>`, December 1996

[14] Robert Mühlbacher and Gustaf Neumann: *Towards a Framework for Collaborative Software Development of Business Application System*, Proceedings of the Fifth Workshops of WET ICE 96, Stanford, 1996

[15] Netscape Communications Corp.: *Plug-In Guide*, `<http://developer.netscape.com/library/documen` May 1997

[16] Netscape Communications Corp.: *JavaScript Reference*, `<http://developer.netscape.com/library/do` October 1997

[17] Gustaf Neumann, Stefan Nusser: *Wafe - An X Toolkit Based Frontend for Application Programs in Various Programming Languages*, USENIX Winter 1993 Technical Conference, San Diego, January 1993.

[18] Object Management Group: *The Common Object Request Broker: Architecture and Specification*, `<ftp://ftp.omg.org/pub/docs/formal/97-10-01.pdf>`, August 1997

[19] John K. Ousterhout: *Tcl: An embeddable Command Language*, Proceeding USENIX Winter Conference, January 1990.

[20] Dave Raggett: *HTML 3.2 Reference Specification*, W3C Recommendation, `<http://www.w3.org/TR/REC-html32.html>`, January 1997

[21] David Wetherall and Christopher J. Lindblad: *Extending Tcl for Dynamic Object-Oriented Programming*, Proceedings of the Tcl/Tk Workshop '95, Toronto, July 1995.