FROM DOCUMENTS TO USER INTERFACES
UNIVERSAL DESIGN AND THE EMERGENCE OF ABSTRACTION

Jason White

ABSTRACT

Abstract representations of content which allow it to be automatically adapted to suit the delivery context, have emerged historically with the development of markup languages intended to facilitate the storage and processing of electronic documents. This technological tradition is reviewed in the first part of the paper, focusing predominantly on the nature and advantages of a 'single authoring' approach to the creation of content. Some of the lessons to be derived from the evolution and deployment of markup systems are also discussed, then applied, in the second part of the paper, to the question of how such abstractions can be extended to the design of user interfaces. Innovative work related to the generic specification of user interfaces is reviewed. It is argued that the advantages of an abstract approach depend for their realization on the development of more expressive style languages and more sophisticated adaptation mechanisms, as well as continued refinement of the semantics of markup languages themselves.

INTRODUCTION

In a paper presented at the Seventh International World Wide Web Conference in 1997 [Vanderheiden, 1997], Gregg Vanderheiden envisioned a technological future in which software applications and informational resources would be universally available, irrespective of the needs and circumstances of the user. Whether behind a desk, in a vehicle, in a noisy environment or a remote location, to mention only some of the diverse possibilities, the same content and applications would be accessible, via whatever hardware and software the circumstances demanded. Leaving aside the important but typically overlooked question of whether such ubiquitous network connectivity is socially desirable, this paper examines some of the basic technologies required to implement Vanderheiden's conception, reviewing the progress that has been made to date, and identifying problems that remain to be solved. Not surprisingly, given the scope of this undertaking, the analysis can neither be detailed nor comprehensive. What is offered, rather, is a general overview of principles and technologies, suitably supported by examples drawn from languages and software systems with which the author is acquainted. The technical coverage offered is likewise diverse, touching on issues of general design as well as more practical questions of deployment and availability.

MARKUP LANGUAGES AND DOCUMENT PROCESSING

The concept of markup has its origin in typesetting. By annotating or 'marking up' a manuscript, a designer would specify to a compositor precisely how the published document was to be formatted and presented. Following the transition from manual to electronic printing, the typesetting activity itself, that is the construction of an image of the page which could be reproduced on paper, was now carried out not by human beings, but by a computer program. Even so, the former division of labour between designer and compositor remained; the files supplied as input to the electronic typesetting system would include, in addition to the text of the document, control sequences to direct the typesetting algorithms. This electronic markup, clearly analogous to its manual predecessor, provides the historical and conceptual base from which modern markup languages have been formed. The subsequent development of markup systems has taken two distinct turns of central interest for the purposes of the present discussion: first, the move from concrete procedural semantics toward more abstract structural descriptions of the document, and secondly, the emergence of syntactically defined types of which document instances are specific manifestations.

By the first of these moves, instead of specifying the font changes, spacing and other characteristics of the manuscript throughout the file wherever they occurred, the structure of the document was represented at a higher level, leaving the details of the formatting to be supplied elsewhere, for example in a macro package. Low-level typesetting operations were superseded, in the markup, by a more general description of the document, distinguishing headings, paragraphs, lists, footnotes, table structures and other components of the material. Each instruction in the markup would be executed by a call to the corresponding macro containing the procedures to be performed in generating a properly typeset image of the page. Of course, this description is something of an idealization; in practice, documents prepared in macro-based typesetting languages such as Troff and TEX [Knuth, 1986] are usually an admixture of structural markup and lower level commands that directly control the presentation. Nevertheless, having introduced a distinction between a declarative, structural description of the document, and the operations and parameters needed for its realization by a typesetting system, the conditions were laid for a more radical move, the conception of an electronic document as instantiating a formally defined grammar, a so-called 'document type', to use the terminology of the Standard Generalized Markup Language (SGML) [International Organization for Standardization, 1986], and its simplified but more popular successor, the Extensible Markup Language (XML) [Bray et al., 2004]. A document type definition (DTD), or more recently, an XML schema [Fallside, 2001, Thompson et al., 2001, Biron and Malhotra, 2001], specifies, most importantly, the names and possible orderings of the elements that can comprise documents of the defined type. Textual content in the form of character strings, may also be permitted to occur in contexts prescribed by the document type definition; likewise, elements may carry attributes, the names and contents of which are again determined by the formal grammar. A principal function of an SGML or validating XML parser is to determine whether a supplied document instance is of the declared type, namely whether it fulfills the requirements set out in the DTD or schema.

By providing a grammar with which to specify document types, SGML and XML facilitate the development of multiple, inter-operating application programs capable of processing document instances. No longer is the markup in the document tied directly to macro calls; instead it is read, parsed and processed by software, the details of which remain entirely unspecified by the markup language, and which may access the content for purposes other than typesetting. Coordinate with these developments, standardized style languages have been created for associating elements of the document with rules, possibly in a quite contextually dependent fashion, that control the typesetting process [International Organization for Standardization , 1996, Bos et al., 1998, Adler et al., 2001]. These rules are generally stored in style sheets, files that are read by the formatting software along with the document instance and any other associated resources, such as vector or raster-based images, required to construct the final presentation. Each style sheet defines a transformation which allows documents conforming to a particular DTD or schema to be presented as determined by the style sheet itself, as applied and executed by the typesetting application. Any number of alternative style sheets, or indeed application programs, may be written that correspond to a particular document type, allowing document instances of that type to be processed in multifarious ways. It is this flexibility which enables a single document instance, in principle, to be formatted for a variety of visual, auditory and tactile displays, each with its own constraints and stylistic requirements, without ever modifying the markup of the document itself. In so far as document instances satisfy formally defined type definitions, the software or style sheet developer can proceed with the assurance that document instances will exhibit a tightly constrained and predictable structure, which, moreover, is specified in a formal language, thereby easing the construction not only of general tools for creating and reading marked up documents, but of highly customized applications and processes for treating documents of particular types.

The advantages of the model that has thus been outlined, in which documents are written in formally defined markup languages, have already been indicated. Nevertheless, in order that certain points can be elaborated further, there is value to be gained by identifying them more precisely. In contrast to

procedurally oriented markup systems, a greater separation is maintained between the electronic representation of the document, and the types of processing and transformation to which it may be subjected. This is achieved by abstraction: the markup typically identifies aspects of the document which the designer of the document type, having more or less regard to the various purposes for which it may be used, considers structurally and semantically significant. With application software of sufficient flexibility and sophistication, the document can be rendered to suit whatever output device, presentational needs and preferences the user may possess. The conversion of an abstractly marked up document instance into a concrete presentation can be carried out entirely by software operating on behalf of the author or the user; or the rendering procedure may alternatively be separated into distinct transformational stages, each carried out in succession at different points along the delivery path from the author's input to the perceived presentation provided by the output device, for example a printer, a video monitor, a tactile display or a speech synthesizer. The characteristics of the hardware and software through which the user interacts with the document, along with those of the network by which it is delivered and the user's needs and preferences, are collectively known as the 'delivery context' [Gimson, 2003]. With this terminology, it can be said that an abstractly marked up document can automatically be adapted to, and presented in, a wide range of delivery contexts, subject to the availability of appropriate software and assuming that the markup specifies explicitly the necessary semantic distinctions to enable the required processing to be carried out. As a corollary, the presentation of the document can be systematically changed merely by altering the rules governing its presentation, for example a style sheet, without changing the document instance. This advantage, in its own right, has often been cited as a sufficient ground for deploying structured markup systems, even in contexts in which the adaptation of documents to a multiplicity of delivery contexts is not considered important. Moreover, the declarative semantics of the markup enable document corpora to be searched more intelligently, with proximity queries limited to specified structural components of the documents, and techniques for ranking search results that rely on this inherent structure to gauge their relevance to the user's interests.

Along with these considerable benefits come, not surprisingly, issues and problems that remain to be addressed. Thus, style languages have not, in general, acquired the degree of expressive power to be found in macro-oriented typesetting languages such as TEX and its associated tools. In practice on the World Wide Web, where markup languages have been most broadly and spectacularly deployed, the emergence of style languages has been gradual, and sophisticated style languages such as XSLT [Clark, 1999], capable of achieving relatively complex hierarchical transformations, remain a rarity, except to some extent in server applications. Likewise, markup languages for representing fonts and vector graphics are far from common on the Web, whereas languages such as Postscript [Adobe Systems, 1999] lie at the core of modern versions of TEX and other typesetting systems. Similarly, to mention a further example, low-level markup languages designed to capture the prosodic parameters of a speech synthesizer by which document can be formatted and rendered in audio, have only been standardized quite recently [Burnett et al., 2004], and their degree of practical implementation has so far been minimal.

Compounding these difficulties has been a tendency, which is again most prominent on the Web, to implement document markup languages such as HTML [Raggett et al., 1999] directly in the compiled code of the user agent software, rather than in a style language. Consequently, support for a new markup language, or even, in many circumstances, an extension of or update to an already implemented markup language, can often only be achieved by providing compiled software that must be downloaded and installed by the user to be executed by the operating system. This compares unfavourably with a design in which core presentational functions are implemented in the user agent, with support for semantically rich markup languages being provided by interpreted or compiled code written in a high-level programming language, which can be downloaded dynamically, cached and executed in a secure environment established by the user agent.

The relative permissiveness of HTML user agents with regard to document instances that fail to conform to a document type definition, combined with the predominance of the graphical desktop workstation as a delivery context, has contributed to the common practice of abusing features of the markup language to achieve a desired layout or other presentational effect, instead of adhering to the semantics defined in the specification. The practical difficulty mentioned earlier of implementing and deploying an extension to a markup language widely supported in user agents, has only exacerbated these problems, leading ultimately to a vast legacy of syntactically invalid and presentationally-oriented content. Both of these aspects are problematic: in so far as the documents are syntactically invalid and unpredictable, the difficulty of writing software that can successfully parse and process them is amplified; in as much as the documents fail to conform to the semantics prescribed for elements and attributes of the markup language, they are less amenable to automatic adaptation for presentation in diverse delivery contexts.

Two lines of advance toward solving these problems are already implicit in the preceding analysis. First, there is a need to develop more sophisticated document presentation systems, encompassing scalable rendering functionality suitable to a variety of devices and output modalities, and high-level style languages for generating optimal presentations of content in a range of delivery contexts. Secondly, and of particular advantage to legacy user agents, there is the possibility of shifting more of the transformation and adaptation process from client to server, whether the latter be controlled by the original author of the document or a third party. In recent years, the latter approach has received considerable impetus from the emergence of portable computing devices and the concomitant need for Web documents to be adapted to the comparatively small displays and limited functionality offered by this hardware.

APPLICATION OF MARKUP LANGUAGES TO USER INTERFACES

In the preceding section, the focus of the analysis was deliberately confined to the adaptation and presentation of electronic documents, setting aside altogether problems related to interactivity. Thus the question arises whether the approach championed earlier in the paper, whereby content is written in abstractly specified and syntactically well defined markup languages, then transformed according to separately provided rules to suit the requirements of the delivery context, can be applied to the construction of user interfaces in general. Promising work has recently been carried out in extending the basic approach to interfaces which, unlike static documents, accept input from the user and undergo modification, whether in response to input or other changes in the state of a program, as application software is executed. Two related specifications encapsulate the results of this effort: the XML Forms Language (XForms) [Dubinko et al., 2003], and the draft Protocols for a Universal Remote Console currently under development by the INCITS V2 working group [InterNational Committee for Information Technology Standards , 2004a,b,c,d,e]. The overview which shall be offered here is necessarily partial; its purpose is to expound the common principles and abstractive techniques underlying the design of both specifications, while largely disregarding important differences between them. To that extent, vital technical details are omitted altogether, or mentioned only incidentally. For the sake of clarity, the terminology of the draft Universal Remote Console specifications, which is more widely applicable to a variety of user interfaces, shall here be used, except where the XForms language is specifically under discussion; but this should not be taken to imply any claim that either design is technically superior to, or more general than, the other.

With these reservations firmly established, the basic components of an abstractly represented user interface can now be identified. The purpose of a user interface is to enable the user of an application to enter input, call software functions which process it, and render the output. Accordingly, the connection between the user interface and the data processing components of the application consists, first, of the constants and variables available to be presented to, and in the latter case set by, the user; and secondly, the application functions that the user can directly invoke. In the Universal

Remote Console specifications, these components are collected in what is known as a user interface socket—an XML document in which variables, constants and application commands are declared. In XForms, the same purpose is served by the XForms model, combined with XML events [McCarron et al., 2003] to enable application functions to be activated. In both cases, the data type of each variable can be constrained by XML schema declarations [Biron and Malhotra, 2001], and XPath expressions [Clark and DeRose, 1999] can be supplied to ensure that validity requirements encompassing one or more variables are met prior to the processing of the data.

Having thus described the data and functionality accessible via the user interface, it is necessary to characterize the actions that the user can perform to set or update the values of variables and to call application functions, together with the outputs that may be presented in response. This is achieved by defining elements referred to in the Universal Remote Console specifications as abstract interactors, which in turn are based on the corresponding XForms form controls. As the name implies, the abstract interactors do not presuppose any particular style of presentation or input mechanism. They are equally suited, for example, to the construction of a speech-based interface oriented toward the processing of natural language, as to the creation of a graphical user interface. Each abstract interactor is explicitly bound to an element of the user interface socket, subject to limitations imposed by its type. Each type of interactor, represented as an XML element, supports a specific operation within the user interface: selection of a single item from a set, selection of multiple items from a set, selection of a value from a totally ordered set, text input, the presentation of output, and execution of application functions, among others [for an early development of these user interface abstractions see Raman, 1997]. The data entered must of course satisfy the requisite type and validity constraints defined in the user interface socket. Abstract interactors may be arranged in logical hierarchies reflecting the structure of the user interface, with related options and actions being grouped together. For example, in the user interface of a bibliographic database, the input fields into which the title, author and other details of the search query are entered, form a logically distinct group, separate from the options that place limits on the number of records to be returned or control the order in which matching records are to be sorted.

Even an abstractly defined user interface is not complete without labels and explanatory text, written in one or more natural languages, that identify and help the user to understand the options provided. In XForms, this essential information is included as part of each form control. In the Universal Remote Console specifications as presently drafted, however, it is stored separately in resource sheets that may be retrieved and interpreted as part of the process of deriving a concrete user interface from the abstract description. This allows for a multiplicity of resource sheets corresponding to any given application, each customized to suit the natural language, culture and other characteristics of a given class of users. In addition to associating a label and help text with an abstract interactor, a resource sheet can specify key words that identify the interactor; these may be used, for instance, by speech recognition systems. Correspondingly, a single character can be bound to the interactor to enhance operation via a keyboard or similar input device. Reference to modality-specific resources such as visual and auditory icons, or formal grammars for speech recognition systems, is also supported. It is the task of the software implementing the universal remote console to combine the abstract definition of the user interface with the data from an applicable resource sheet to create an interface suited to the requirements of the delivery context.

Just as generic markup has advantages over procedural markup in that it separates the details of layout and formatting from the structural description of a document, thus permitting a document instance to be processed in ways that are limited only by the ingenuity of the application or style sheet developer, it has the potential to yield corresponding benefits as applied to the abstract characterization of user interfaces. Most user interfaces designed today are written in a conventional programming language, with a quite specific delivery context in mind, for example a graphical workstation, and are carefully designed for the intended environment. Other delivery options are, at best, treated as secondary concerns: a user interface library may, for instance, support accessibility

via assistive technologies that provide alternative input and output mechanisms for people with disabilities, but the graphical interface remains the primary focus of the design. What distinguishes XForms and the Universal Remote Console specifications is the achievement of a more adequate demarcation of the underlying structure from those aspects of the user interface which vary with the delivery context, while allowing additional resources to be specified that enable the interface to be automatically enhanced or customized for specific environments. Furthermore, this generic approach challenges the traditional distinction between user interfaces as such, intended to be operated directly by human beings, and application interfaces exposed to external software. Thus one can envisage complex and largely automated tools that locate and interact with devices and applications supporting such interfaces, with little, if any, direct human intervention. It is these possibilities that open the way to the prospect of sophisticated systems composed of collaborating, networked devices serving our needs through interactive and evolving interfaces that take advantage of whatever input or output mechanism the occasion demands [see the discussion in Vanderheiden and Zimmermann, 2004]. In short, it is partly through making the user interface amenable to automated as well as human interaction that the technological future which Vanderheiden imagines can become actualized.

Although, as XForms and the Universal Remote Console specifications illustrate, significant progress has been made in recent years toward the development of suitable abstractions with which to describe user interfaces generically, the problem of transforming these into concrete interfaces optimized for the delivery context, remains still to be solved. As has already been pointed out, in the application of generalized markup to electronic typesetting, the rules relating the structure of a document to its presentation in the final output, have come to be codified in style languages that control the rendering process. Given a document instance conforming to a specified schema and an appropriate style sheet, a typesetting system can generate a quality presentation tailored to the delivery context for which the style rules are designed. The convenience of this solution lies as much in the availability of flexible style languages with which to specify presentation, as in the abstractions underlying the markup with which the structure of the document is captured. To gain the benefits of extending this basic model to the design of user interfaces, it is desirable to develop versatile and sophisticated style languages with which to express general rules whereby abstract descriptions can be converted into highly usable, effective and consistent interfaces, with each adapted to a particular delivery context. By thus partially automating the derivation of concrete interfaces from abstract specifications, it would become possible, as in the field of document processing, to apply a common set of style rules to a multiplicity of interfaces, making only such modifications as are needed to accommodate the unusual or exceptional features of a specific software application. The advantages noted earlier in connection with document processing, including consistency of presentation and the opportunity to make uniform changes simply by modifying applicable style rules, would carry over, mutatis mutandis, to the construction of user interfaces. Thus, the predictability of interactional conventions within and among applications would be enhanced, without limiting the flexibility of the software developer, in writing specially crafted style rules, to customize the interface for chosen delivery contexts. Adaptation of an existing abstract interface to a new delivery context could be accomplished by defining a new set of style rules, or, in more complex situations, implementing specialized software for interpreting the markup in which the interface is described. As is true of markup systems generally, languages for characterizing abstract user interfaces can be interpreted in a server or client environment according to the requirements of the implementation, including the extent to which compatibility with legacy software and standards is necessary[this point is further elaborated in Maes and Raman, 2001].

CONCLUSIONS: THE CHALLENGE FOR USER INTERFACE DESIGN

Structured markup languages have here been considered in connection with two domains of application: the representation of electronic documents, and of interactive user interfaces, respectively. These should not indeed be regarded as mutually exclusive categories; rather, the latter introduces a dynamic dimension that is absent from the former. A conventional document can thus be conceived of as a purely presentational, entirely non-interactive, user interface. It has been argued in this paper that the use of generic, syntactically defined, markup languages for the encoding of documents, is a technically advantageous practice in that it allows the abstract representation of the electronic content to be kept distinct from the various means by which it may be processed and displayed to a user. An important benefit of this design lies in the flexibility with which structurally marked up documents can be adapted, typically by applying style rules, to a wide range of delivery contexts. Significant shortcomings in contemporary implementations of the preferred approach have also been noted, including a failure to build consistently upon well designed foundations provided for example by vector graphics and audio formatting languages [the concept of an audio formatting language is developed in Raman, 1998], and the inadequacy of currently deployed style languages by contrast with typesetting systems such as TEX. These limitations have also given rise to difficulties in deploying extensions to popular markup languages, in particular HTML.

The demands placed on the design of style languages and other mechanisms that process, transform and render marked up content, become all the more complex if what is abstractly represented is not merely the structure of a static document, but instead that of a dynamic user interface. As recent work in the development of markup languages for describing user interfaces has shown, appropriate abstractions enable the essential aspects of an interface to be captured quite independently of any assumptions regarding the delivery contexts in which it may be presented. The technologies needed to adapt such abstract interfaces to various delivery contexts, and, where possible, to do so by applying rules that can be conveniently specified by the application designer or by a third party, are yet to be developed. To what extent the craft of user interface design, like that of typography, can be carried out by specifying rules for converting abstract structural descriptions into concrete presentations and interactions optimized for the delivery context, continues to be largely an unanswered question. In typesetting and document processing, there is already a strong argument for the use of generic markup systems[see also Taylor, 1996]. Many of the advantages of this technology which have been identified with respect to electronic documents, have counterparts, so it has been argued, in its application to user interfaces, assuming of course that the required processing and adaptation mechanisms, as already mentioned, can in fact be developed. In so far as these benefits are realized, the abstract solution may largely supersede the current practice of individually coding a user interface specifically for its intended delivery context. The technological possibilities which this would open up, and the benefits that would thereby accrue to users, have barely been touched on in this paper. However, the broad advantages have been clearly identified, leaving the detailed possibilities to be further explored as the technologies themselves are developed.

REFERENCES

Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. **Extensible stylesheet language (xsl) version 1.0. Recommendation,** W3C, October 2001. URL http://www.w3.org/TR/2001/REC-xsl-20011015/.

Adobe Systems. **Postscript Language Reference**. Addison-Wesley Pub. Co., third edition, 1999.

Paul V. Biron and Ashok Malhotra. Xml schema part 2: Datatypes. Recommendation, W3C, May 2001. URL http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2. Recommendation, W3C, May 1998. URL http://www.w3.org/TR/1998/REC-CSS2-19980512.

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (xml) 1.1. Recommendation, W3C, February 2004. URL http://www.w3.org/TR/xml11.

Daniel C. Burnett, Mark R. Walker, and Andrew Hunt. Speech synthesis markup language (ssml) version 1.0. Recommendation, W3C, September 2004. URL http://www.w3.org/TR/2004/REC-speech-synthesis-20040907/.

James Clark. Xsl transformations (xslt) version 1.0. Recommendation, W3C, November 1999. URL http://www.w3.org/TR/1999/REC-xslt-19991116.

James Clark and Steve DeRose. Xml path language (xpath) version 1.0. Recommendation, W3C, November 1999. URL http://www.w3.org/TR/1999/REC-xpath-19991116.

Micah Dubinko, Leigh L. Klotz, Jr., Roland Merrick, and T. V. Raman. Xforms 1.0. Recommendation, W3C, October 2003. URL http://www.w3.org/TR/2003/REC-xforms-20031014/.

David C. Fallside. Xml schema part 0: Primer. Recommendation, W3C, May 2001. URL http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

Roger Gimson. **Device independence principles**. Note, W3C, September 2003. URL http://www.w3.org/TR/di-princ/.

InterNational Committee for Information Technology Standards . Ansi/incits 389, information technology—protocol to facilitate operation of information and electronic products through remote and alternative interfaces and intelligent agents: Universal remote console, 2004a.

InterNational Committee for Information Technology Standards . Ansi/incits 390, information technology—protocol to facilitate operation of information and electronic products through remote and alternative interfaces and intelligent agents: User interface socket description, 2004b.

InterNational Committee for Information Technology Standards . Ansi/incits 391, information technology—protocol to facilitate operation of information and electronic products through remote and alternative interfaces and intelligent agents: Presentation templates, 2004c.

InterNational Committee for Information Technology Standards . Ansi/incits 392, information technology—protocol to facilitate operation of information and electronic products through remote and alternative interfaces and intelligent agents: Target properties sheet, 2004d.

InterNational Committee for Information Technology Standards . Ansi/incits 393, information technology—protocol to facilitate operation of information and electronic products through remote and alternative interfaces and intelligent agents: Resource description, 2004e.

International Organization for Standardization . Iso 10179: Document style semantics and specification language, 1996.

International Organization for Standardization. Iso 8879: Standard generalized markup language, 1986.

Donald E. Knuth. **The TeXbook**. Addison-Wesley Pub. Co., 1986.

Stéphane H. Maes and T. V. Raman. A “single authoring” programming model for the next web. In Scientific Emphasis Proceedings, 2001. URL http://www.stephanemaes.com/ESSEM/ScientificEmphasis/Proceedings/2001/06/singleauthoring.htm.

Shane McCarron, Steven Pemberton, and T. V. Raman. Xml events: An events syntax for xml. Recommendation, W3C, October 2003. URL http://www.w3.org/TR/2003/REC-xml-events-20031014.

Dave Raggett, Arnaud Le Hors, and Ian Jacobs. Html 4.01 specification. Recommendation, W3C, December 1999. URL http://www.w3.org/TR/html401.

T. V. Raman. **Auditory User Interfaces: Toward the Speaking Computer**. Kluwer Academic Publishers, 1997.

T. V. Raman. **Audio System for Technical Readings**, volume 0302-9743; 1410 of Lecture Notes in Computer Science. Springer-Verlag, 1998.

Conrad Taylor. **What has wysiwyg done to us?** The Seybold Report on Publishing Systems, 26(2), September 1996.

Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Xml schema part 1: Structures. Recommendation, W3C, May 2001. URL http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.

Gregg C. Vanderheiden. Anywhere, anytime (+anyone) access to the next-generation www. In Proceedings of the Sixth International World Wide Web Conference, 1997. URL http://decweb.ethz.ch/WWW6/Technical/Paper253/Paper253.html.

Gregg C. Vanderheiden and Gottfried Zimmermann. Interface sockets, remote consoles, and natural language agents: A v2 urc standards whitepaper, August 2004. URL http://www.incits.org/tc_home/V2HTM/docs/V2/04-0065/V2Whitepaper.htm.