Application of Information Technology ■

WebEAV:

Automatic Metadata-driven Generation of Web Interfaces to Entity-Attribute-Value Databases

PRAKASH M. NADKARNI, MD, CYNTHIA M. BRANDT, MD, LUIS MARENCO, MD

Abstract The task of creating and maintaining a front end to a large institutional entity-attribute-value (EAV) database can be cumbersome when using traditional client–server technology. Switching to Web technology as a delivery vehicle solves some of these problems but introduces others. In particular, Web development environments tend to be primitive, and many features that client–server developers take for granted are missing. WebEAV is a generic framework for Web development that is intended to streamline the process of Web application development for databases having a significant EAV component. It also addresses some challenging user interface issues that arise when any complex system is created. The authors describe the architecture of WebEAV and provide an overview of its features with suitable examples.

■ **J Am Med Inform Assoc.** 2000;7:343–356.

The entity-attribute-value (EAV) physical database architecture is widely used in clinical data repositories (CDRs). Those CDRs with a major EAV component include the pioneering HELP system[1,2] (and its commercial version, the 3M CDR[3]) and the Columbia–Presbyterian Medical Center CDR.[4,5] Entity-attribute-value design addresses a problem that conventional table design (i.e., one column per finding or parameter) cannot address. Specifically, data on several thousand potential parameters can be stored for a patient across all clinical specialties. If these data are modeled as one database field per parameter, numerous tables are required to hold the data, and these tables will require repeated modification as medicine advances and new clinical and laboratory parameters need to be recorded. Searching across numerous tables for all data on a single patient is also inefficient, especially for the vast majority of patients, for whom only a modest number of parameters are actually applicable.

In EAV design, we have (conceptually) a single table that records the data as one row per finding. Each row contains the following information: **entity** (patient identification, visit, date/time etc.), **attribute** (the name/identification of the parameter), and the **value** of the parameter. Because attributes are not hard-coded as database fields, this design does not require revision as new clinical parameters enter the medical domain. Data retrieval is also efficient. To retrieve all the facts on a patient, one simply searches the entity column(s) for the patient identification, ordering all rows by date and time if necessary.

Although EAV architecture dramatically simplifies CDR database design, it complicates user interface design significantly. Specifically, the **global schema** of an EAV database (the way the data are actually organized into tables) differs greatly from its **logical schema** (the way they are perceived as being organized). In our experience, end users tend to regard the data as being stored conventionally as one database field per attribute, even if they are not. Almost all analytic programs, such as spreadsheet or statistics packages, also expect input data to be organized conventionally. Therefore, CDR system architects must spend considerable effort simulating the logical schema, especially when presenting data on a particular patient through forms or data entry screens. This

Affiliation of the authors: Yale University School of Medicine, New Haven, Connecticut.

This work was supported by grant U01-CA-78266 from the National Cancer Institute.

Correspondence and reprints: Prakash M. Nadkarni, MD, Center for Medical Informatics, Yale University School of Medicine, P.O. Box 208009, New Haven, CT 06520-8009;
e-mail: ⟨prakash.nadkarni@yale.edu⟩.

Received for publication: 10/1/99; accepted for publication: 2/2/00.

means converting EAV data to a conventional structure before they are presented to the user and translating the edited data back into EAV structure when edits are to be posted back to the server.

There are currently several powerful front-end tools for client–server database development. These include desktop database management systems such as Microsoft Access, Paradox, and Visual Foxpro, as well as programs such as Visual Basic and PowerBuilder. Although such tools often make it possible to create forms without programming, their form design facilities are based on the one-record-per-screen metaphor. They work well for conventional but not for EAV data, where the data on one *logical* record (e.g., all findings pertaining to a single patient event) are stored as multiple *physical* records, with one record per finding. Provision of form interfaces for EAV databases therefore requires much custom programming.

This paper describes WebEAV, a Web-oriented programming framework that minimizes the amount of programming by permitting the automatic generation of Web-based forms for input and display of EAV data. The forms that are generated provide a robust set of features and functionality. We are using WebEAV for two production EAV databases:

- ACT/DB, a database system for managing clinical studies data.[6,7] This is in multidepartment production use at Yale and at the Vanderbilt University Cancer Center. It is also the basis for a special studies database for the U.S. National Cancer Institute–supported Cancer Genetics Network initiative.[8]
- SENSELAB,[9,10] a collection of heterogeneous neuroscience data (sequences, neuronal models, circuits, experiments, etc.) centered on the olfactory system.

Background

Developing Traditional Front Ends for EAV Data

In this section, we discuss the significant maintenance problems that arise when traditional client–server approaches are used to create front ends for complex, multi-user EAV databases. We first describe two approaches for browsing and editing EAV data that are based on mapping sets of attributes to tables that reside on the client and are managed by a desktop database management system. These tables transiently capture data from the server, and the user manipulates the data through forms based on these tables.

The availability of client-side tables greatly simplifies presentation of data that have many-to-one relationships, where the ''many'' records are to be displayed simultaneously with the ''one'' record. For example, a physician inspecting a cancer patient's demographic data may also wish to see details of multiple past episodes of surgery or radiotherapy. Traditional client–server systems handle these presentation needs by letting the developer create *subforms*, one or more of which can be embedded in the main form. Thus, surgery and radiotherapy data, which are also transiently captured in their own tables, are displayed in separate subforms within the demographics form. As discussed later, simulating subforms that also permit data entry and editing on the Web requires complicated programming.

Static Table-based Mapping

In static mapping, each client table reflects an individual data collection instrument. This is a paper-based or electronic form used to gather or present data on a set of related clinical parameters, e.g., routine hematology or a standard clinical chemistry panel such as the SMA-14. Each field on a particular form (e.g., the field ''Hemoglobin'') is mapped to a counterpart in the EAV schema (the attribute ID for hemoglobin, e.g., 1135). For such purposes, most traditional client–server environments provide a ''tag'' for every object in a form. The tag can contain arbitrary developer-assigned text whose interpretation is left to program code; thus, the attribute ID can be stored in the tag. The drawbacks of this mapping approach are as follows.

- A large system may require several hundred tables, each with associated forms. The forms and tables can take up considerable space on a client machine, even if the tables are generally empty. It is possible to save space by storing, on clients within individual departments, only those tables and forms that the department needs to use. Maintenance of department-specific sets of tables, however, constitutes significant administrative and manpower overhead.
- Revisions and bug fixes to tables and forms require the corresponding tables or forms to be reinstalled on individual machines. In our experience, many nonstandard data collection instruments that are being devised for brand-new protocols change repeatedly—often four or more times—as investigators iteratively converge on a decision regarding the set of parameters to be gathered.
- The number of form-entry fields per form is limited to the maximum number of database fields per table (typically, 255), which may be limiting for large

data collection instruments (e.g., certain psychiatry questionnaires such as personality inventories). One must then use inelegant workarounds such as splitting up a large data collection instrument into two or more separate forms based on separate tables.

Dynamic Table-based Mapping

Dynamic table-based mapping, a significant improvement over static mapping, was implemented in an earlier version of ACT/DB. Here, the client has a few general-purpose, *reusable* tables with numerous database fields whose mapping to attributes in the EAV schema can change, depending on the form being displayed. That is, these tables are used to transiently capture all EAV, irrespective of the DCI.

One table is used to display data in the main form. The number of subform tables required depends on the maximum number of subforms per form across the system. In our experience, five or six reusable subform tables generally suffice.

The fields in such tables are named serially. In ACT/DB, which uses strong data typing, one designates such fields to hold strings, integers, decimal numbers, dates, and so forth. Thus, the first database field for string data would be named "S01." Strong typing greatly reduces the programming needed for client-side data validation. For example, the built-in validation facilities of many traditional client–server environments prevent entry of alphabets in numeric fields, and date fields reject invalid dates, even handling leap-year logic correctly. "Pictures," which are templates to restrict data entry, such as "(999) 999-9999" for phone numbers, also assist validation and data standardization.

The server's metadata ("data dictionary") records, for every form, the mapping of specific database fields to their corresponding form fields. Subsets of the mapping metadata are replicated programmatically on demand on individual clients, on the basis of the forms that each client uses. When a particular form is about to be opened, its mapping metadata are refreshed from the server if the latter are more recent, as determined by a comparison of time stamps. If the new metadata are incompatible with the old (e.g., the number of fields for one or more data types has changed), the user can be warned that the form on the client is obsolete.

Depending on the client software and setup, it may or may not be possible to automatically download the current version of the form from a "forms server." With Microsoft Access, for example, form transfer without workflow interruption requires clients to have a full version of Access installed. If, however, clients are using Access Runtime (which allows unrestricted application distribution, without per-machine licensing costs), this is not possible, because all forms are treated as having been "compiled" into the application.

By using other metadata—such as the data type and brief description of attributes, the order in which they are to appear in the form, and their aggregation into logical groups—it is possible to write a code library to generate forms, and their mapping metadata, automatically. ACT/DB and SENSELAB both contain such a library, which is described by Nadkarni et al.[11]

Dynamic table-based mapping solves the table proliferation problem, but the maximum-fields-per-form limitation remains. The limit may in fact be reached sooner than with static mapping; for example, all the available string or integer database fields may be used up during the creation of a large form, even if most of the date fields are unused. The dynamic mapping approach also fails to fully address the forms-maintenance problem, because the existence of an obsolete form, while detected correctly, interrupts workflow if the form must be manually downloaded and reinstalled. Furthermore, with a large form the delay caused by metadata downloading and metadata version checking may be significant.

Form Reuse Issues

Multiple departments may use the same data collection instrument with varying degrees of detail. Thus, in a hematology panel, tracking of peripheral promyelocytes or metamyelocytes may be important for cancer chemotherapy but not for routine screening. If one creates numerous department-specific forms for what is fundamentally the same instrument, forms proliferation becomes hard to manage. Reuse of forms that record the greatest common denominator of information is therefore preferred. However, if a given department is concerned with only 5 parameters on a form that has placeholders for 20, it can confuse users who see many more form-entry fields than are appropriate to their needs. Especially if data are being entered through transcription from paper forms, it is important that the data entry person not be presented with fields that do not exist on the paper form.

ACT/DB addresses this issue by permitting the designer to specify, for a given study, which fields on a given form are *required*. When a particular user opens the form, then, based on the current study, the background color of "required" fields is dynamically set to a pale yellow to indicate which fields on the form can be ignored. This solution is a partial one, because

the form is still busier than it should be; ideally, fields that are not required should simply not be shown. One can write generic code to dynamically make non-required fields invisible, but with traditional client–server front ends, the form does not reformat; unsightly gaps indicate invisible fields. Form esthetics becomes a factor if hardcopy is required. Later in this article we discuss how WebEAV addresses this problem.

Creating Web-based Interfaces

The World Wide Web offers a unique opportunity to simplify database deployment. In typical Web database applications, a user's browser requests data from a remote Web server, which in turn requests them from a database server. (The latter may reside on the same machine as the Web server or on a machine in the same network.) After the database server returns the requested data, the Web server formats it into a Web page (in the form of hypertext markup language, or HTML) and sends it to the browser. Additional "application server" software may be placed between Web and database servers; this includes a transaction monitor for tasks such as pooling of database connections to improve response time and reduce database server load.

The advantages of Web deployment are summarized below:

- Problems of maintaining form versions go away, because all forms reside on a Web server, to be downloaded on demand by a client browser. Changes to a given form are automatically available the next time a Web browser accesses the server.
- Web browsers use extremely clever caching algorithms that the developer can leverage. When a browser visits a particular page on a Web site, its contents are cached on the local machine. During a subsequent visit to the same page, only those components (i.e., the HTML, embedded images, applets, or code libraries) that have changed since the last visit are re-downloaded.
- The HTML page- or form-rendering model is both simpler and smarter than that of traditional client–server environments. By default, the objects on a page automatically reformat whenever the browser window is resized or whenever the user changes the font size. Traditional client–server programmers, in contrast, must devote much effort to physical screen size issues. For fine control, "cascading style sheets,"[12] a standard promulgated by the international Web-related standards body, the World-Wide Web Consortium (http://www.w3.org/), provide a high-level means of document formatting. A "style" is effectively a macro that permits the font, color, positioning, and visibility of HTML segments to be specified in exquisite detail. Formatting attributes can be altered by "client-side" code. (Client-side code is code that is part of the page and runs in the browser. It is typically written using the language JavaScript or VBScript, or both.) Modest coding efforts achieve dramatic changes in screen appearance.
- Web-based solutions result in significantly lower deployment costs. Browsers are given away free, and therefore per-seat client licenses are not necessary.

For these reasons, the Web is increasingly the medium of choice for multi-user application deployment, especially for databases. However, Web database applications that must support data editing and entry are significantly more complex to develop than traditional client–server applications, for several reasons.

Communication between browser and Web server via the HTTP protocol is intrinsically "stateless."[13] That is, once the server has handed a page to the browser, it closes the connection and "forgets" about the client. To maintain state, the developer must store state data either on a Web page (using "hidden" or invisible form fields) or in "cookies," which are text items in attribute-value form that are stored by the user's browser on the local machine.

Web forms require much more custom programming than traditional client–server environments for client-side data validation, because Web form fields are typeless and "pictures" are unavailable. While server-side validation can be done (and should be done anyway), providing an error message after the Web form is submitted (and after a variable delay) can cause user frustration. Satisfactory ergonomics are facilitated by maximal validation at the client browser *before* form submission, through client-side scripting. Ideally, an error message should appear immediately after the user tries to move from the field with erroneous data to the next field.

In our opinion, many Web development tools are much less mature than traditional client–server environments, and the edit–test–debug cycle is greatly lengthened. Currently, for example, simple errors such as undeclared or misspelled variables, which would be trapped at compile/edit time in traditional client–server environments, remain undetected until runtime. Web development environments desperately

need an equivalent of the UNIX *lint* utility,[14] which detects questionable constructs in C code.

Design Objectives

Coding Web forms by hand to support robust data browsing and editing is tedious and error-prone. WebEAV is a framework for simplifying such development. These are its objectives:

- The WebEAV framework should automatically generate forms based on attribute metadata. For efficiency reasons, it is desirable to pregenerate as much of a form as possible, so that most of the form is *static* (i.e., unchanged between consecutive uses). However, the form must also contain *dynamic* components that change the form's behavior on the basis of the currently logged-on user and, in the case of ACT/DB, the current study.
- The esthetics of a program-generated form cannot be 100 per cent satisfactory. It is desirable, therefore, to generate forms that can be customized by (non-programmer) lead users with graphical Web-page editors. Providing form-editing capability enfranchises users and improves user satisfaction while freeing developers for more intellectually challenging tasks.
- The Web forms must be responsive in relatively low-bandwidth situations. Therefore, once a form is downloaded, to-and-fro communication must be minimized. In high-bandwidth traditional client–server applications, in contrast, a client may repeatedly contact the server during data entry, e.g., to populate values in a pull-down menu. A form must therefore contain almost all the scripting code and data, including mapping metadata, needed to function autonomously, until the user submits the form.
- WebEAV should not be limited to managing EAV data alone. Most production EAV databases, including our own, store certain types of homogeneous data, e.g., patient demographics, in conventional form for efficiency purposes.
- The ideas embodied in WebEAV should be sufficiently generic to permit porting to other hardware and software platforms.

Our description of WebEAV in this paper is intended to be comprehensive enough that Web developers in biomedical institutions should be able to derive useful ideas from our work even if they do not intend to inspect or use WebEAV code itself.

System Description

WebEAV is currently implemented on the Windows NT platform. It uses Microsoft Internet Information Server as the Web server and Microsoft Transaction Server as the application server; both are part of the default installation of Windows NT Server version 4.0. It uses Active Server Pages, or ASP (described shortly) for server programming. We use Oracle as the database engine (although none of the code in WebEAV is Oracle-specific). On our test system, the database server resides on the same machine as the Web server, while in our production system it resides on a separate machine.

Choice of Software Platform

WebEAV uses ASP technology on the server end. Originally devised by Microsoft for use on their Web server (Internet Information Server, IIS), ASP is also available through a third-party vendor (Chili!Soft) for non-Microsoft Web servers running on non-Windows platforms.

ASP allows a developer to place programming code (written in a "lightweight" scripting language such as VBScript, Javascript, and PerlScript) at multiple places in a Web page. (HTML itself is only a markup language that specifies formatting, not a programming language.) This page is saved with a special file extension (.asp instead of .html). When a browser requests the page, the Web server first passes the page to an interpreter, which executes each instance of embedded code and generates text that is inserted into the page at one or more points. When the browser receives the page, all server-side code has been removed. On Windows NT, the ASP processor and VBScript and JavaScript interpreters are part of the default NT installation.

ASP is not unique in its approach: PHP (http://www.php.net/) is a popular freeware C/Perl-like language environment for UNIX/Windows NT that works on identical principles. Java Server Pages (JSP),[15] which is also available on a variety of Web servers, works in a similar fashion. (The techniques we describe may be readily adapted to PHP or JSP.) The ASP programming model has both advantages and disadvantages.

- ASP offers somewhat higher development throughput than alternatives such as Common Gateway Interface (CGI) programming. (CGI was the first framework defined for Web programming and is supported on all Web servers.) Many complex aspects of Web programming are encapsulated in rel-

atively easy-to-use high-level objects. Persistent connections between Web and database servers are simple to program.

- Unlike in CGI programming, an ASP page does not need to be generated entirely by code. Using Web-page editors, nonprogrammers can specify page layout, e.g., setting fonts and colors or inserting images. They can similarly customize program-generated ASP pages as long as they take care not to delete code.
- On the downside, currently available ASP scripting languages have somewhat limited features and debugging environments. (VBScript code segments, however, can mostly be developed in a more robust environment, e.g., Visual Basic or Microsoft Access, and subsequently copied and pasted into an ASP page.)

Browser Dependencies

Some advanced features of WebEAV require the use of Microsoft Internet Explorer (MSIE) version 5 (the present version). Although we are not happy to limit the user's choice of browser, the versions of dynamic HTML (DHTML) implemented by MSIE and Netscape Navigator (NN) are mostly incompatible, and NN has fallen greatly behind in programmability. (DHTML is the browser-specific programming framework, consisting of objects and methods, through which the contents of an HTML page can be manipulated on the client.) To quote Web development Guru Danny Goodman[16]:

> Literally every HTML element (in MSIE) is exposed as a scriptable object . . . [MSIE] automatically reflows the page whenever you do anything that changes its contents, such as adjusting the font size for a phrase or inserting some HTML text in the middle of the paragraph. . . . Even if the [NN] object model allowed content modification on the fly (which it does not), pages do not automatically reflow in Navigator 4.

For these reasons, MSIE has become the browser of choice for intranet application development.

While supporting all browsers may be politically correct, a developer must seriously consider the cost in terms of programming resources and the restricted functionality deliverable with a least-common-denominator approach. Since both NN and MSIE are freely downloadable, we have taken the view that mandating MSIE—at least for ACT/DB, where we have a closed user community—does not impose insuperable hardships.

Architectural Overview

The WebEAV framework consists of the following components:

- Server-side and client-side *code libraries* whose subroutines can be used by the developer during runtime.
- A *metadata* component that is part of the server database. The metadata holds information on the attributes in the system.
- A *forms-generation library*, which generates the forms by consulting the server metadata. After testing, these forms are moved to a production Web server. This library is hosted in a Microsoft Access application.

The developer uses WebEAV to generate forms (which may be customized after creation) and also uses the code library routines. WebEAV, however, is transparent to the end user, who sees only the application that has been built using WebEAV. We now discuss individual features of WebEAV.

Protection Against Unauthorized Access

Databases employing WebEAV use a standard login/password mechanism to restrict access to authorized users, as well as encrypted communication through the standard HTTPS (hypertext transmission protocol, secure) to minimize the risks of electronic eavesdropping. Additional security measures employed by WebEAV are described below.

A common security loophole in many apparently password-protected Web sites is that only access to the starting (''Home'') page is protected. Once a user has moved past this page, subsequent pages can be bookmarked in the browser. The user can then go directly to these pages without a password. This is a security risk on a shared or unattended machine, where an unauthorized user may be able to inspect confidential patient data by using bookmarks or even the browser's ''history'' information on recently visited Web pages. WebEAV has several defenses against this problem.

- Individual Web pages are set to ''expire'' after a suitable interval (e.g., 1/2 to 1 hour), so that the browser removes these pages from its cache.
- WebEAV server code uses the HTTP_REFERER server variable to determine the URL of the page from which the user navigated to the page about to be displayed. This variable is blank if a user uses

bookmarks or history to jump to this page. Within the application, the user must navigate pages in a particular order; therefore, a given page can have only a limited number of valid "referring" pages. If the Web server detects that navigation to a particular page has bypassed the normal route, the user is redirected to the startup (login) screen.

- A user session receives a properly initialized database "connection object" after successful login. All subsequent Web pages reuse this object for database access for efficiency reasons, rather than creating their own connections. (Connection objects, which can themselves be set to time out appropriately, are *logical* connections—the Web server typically uses connection pooling, where a few *physical* database connections are multiplexed between multiple concurrent users.) If an intruder tries to bypass the login screen by programming HTTP requests that set the HTTP_REFERER variable (rather than relying on a well-behaved browser), an invalid connection object will be received and database operations will fail.

Control of Access for Individual Authorized Users

Individual authorized users have varied access rights with respect to different parts of the logical schema of a database. In ACT/DB, for example, a user who has just logged on should see only the studies to which he or she has access. For a given study, some users can only look at data, while others have editing or administrative privileges.

Since all EAV data are stored in the same set of physical tables, the standard database mechanisms for controlling individual table access (using SQL's GRANT and REVOKE commands) will not work. Therefore, both of our systems maintain user-privilege metadata, which also drive the user interface. Immediately after a user logs on and accesses a study, WebEAV gets the user's privileges for that study and stores them as a bit-string in a hidden field in every Web form. When the user navigates from form to form, privilege information is passed between forms to control the generation of individual user-interface objects and prevent nonpermissible actions. For example, if the user does not have editing privileges, the "Save" button is not generated.

Operation of the Basic Data Collection Instrument Interface

For every instrument, WebEAV generates two forms instead of one. These are presented together as part of a "frame set." The lower, *detail* form contains the attributes whose values will be edited. In the metadata, the developer aggregates attributes into groups, and WebEAV generates an invisible "anchor" against each group's caption. The upper, *header* form contains fields for constantly viewed information, such as patient demographics. For each attribute group, a button is generated with an appropriate caption. Clicking a particular button automatically scrolls that detail form to the corresponding anchor. Buttons and anchors facilitate navigation through a large form.

In ACT/DB, there is also a third form—actually a narrow "toolbar" of buttons—that is placed at the bottom of the frame set. This is shared by all data entry instruments and contains a hidden field that actually records the list of form-entry fields whose contents have been changed by the user during editing, along with their changed values.

A form illustrating header, detail, and toolbar frames, with navigation buttons in the header, is shown in Figure 1.

Handling Mapping Metadata for Data Entry and Database Updates

When changes are made to data presented in a Web form and the form is submitted to the Web server, the server issues one or more SQL UPDATE statements to the database. An individual statement specifies the table to be updated, the primary key values corresponding to the row to be altered, the columns to be altered, and the new value for each column.

Database updating is tedious to implement on a case-by-case basis for several reasons. The data type of each field determines the syntax; thus, string values must be quoted and numeric values unquoted. Furthermore, database vendors generally implement date/time fields idiosyncratically, and portable manipulation of date/times requires custom coding using a standard such as ODBC. It is therefore desirable to create a generic data-updating framework that can be used across all forms in the application. Furthermore, since a Web form, unlike a traditional client–server, environment, does not have the benefit of a local database engine on the client, we must devise a means of addressing the mapping-metadata problem for EAV as well as conventional attributes.

We do this through the technique of *self-mapping field names*. Every data-entry field in a Web form should have a unique name. When WebEAV generates Web forms automatically, it generates field names such that *each field's name is its own metadata*, and is sufficient, after parsing, to generate the correct UPDATE statement.

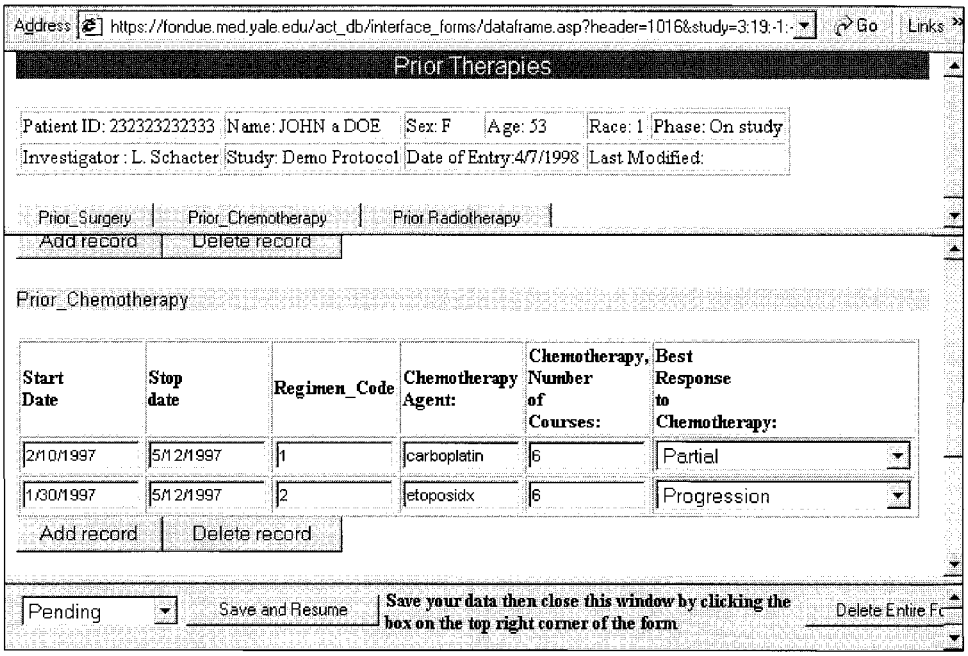

Figure 1 The basic form interface generated for ACT/DB. There are three frames in a "frame set." The top "leader" frame displays demographics data with, optionally, a set of buttons to enable navigation within the second form. The middle "detail" frame is where data browsing and editing are actually done. The bottom "toolbar" frame is shared across all data entry forms: the forms generator generates only the header and detail frames. In the figure, the user has clicked on the "Prior Chemotherapy" button in the header frame, causing the detail frame to scroll to the subform for Prior Chemotherapy.

For fields corresponding to EAV attributes, we synthesize the name by concatenating the *attribute's ID*, a single letter for its *data type* (e.g., S=string, D=Date, etc.), and an *instance ID*. The last is zero if a given attribute occurs only once in a form, and not zero if there are multiple instances, as in a subform. The parts of the field name are separated by underscores for easy parsing.

For fields corresponding to conventional attributes, we concatenate the *column name* in the table where this attribute is stored, with the data type and instance ID. The name of the table, or updatable view, to which the field belongs, is stored in the form as a hidden field with the name "*_TABLENAME_*." Another hidden field, "*_PRIMARYKEYS_*," maintains a comma-delimited list of the primary key field names for the table or view in question.

In MSIE (but not in Netscape), every field on the form —indeed, almost any arbitrary block, or "division," of HTML—can be assigned a string called the *object ID*. The object ID is used to reference the object for client-side manipulation; it is not sent to the Web server after form submission. IDs, despite their name, are not required to be unique; several objects can share the same ID. WebEAV generates IDs for every field, using the attribute ID for EAV attributes and the database column name for conventional attributes. Thus, in a subform, all instances of the same attribute (i.e., elements in the same column) share the same ID. The use of IDs enables computed formulas and skip logic rules (discussed shortly) to be specified just once, even though multiple rows in a subform will each use the same formula or rule.

With self-mapping field names, unlike table-based mapping, the number of fields per form has no artificial limits.

Client-side Data Validation: Use of Events

Client-side Web programming relies on the use of *events*, which are triggered by actions such as depression or clicking of one or more mouse buttons, entry or exit into a field, and change of a field's contents. WebEAV form generation uses the metadata for the attributes in a form (e.g., this description, data type, maximum and minimum bounds, non-null requirements, etc.) to generate standard *event handling code* for the corresponding form fields. WebEAV generates code for several types of events.

- A field's *OnChange* event, which fires when the contents of a field change, calls a standard subroutine that performs type, range, and non-null checking. If the field passes validation, additional code may be invoked as described in the "Advanced features" subsection later.
- The *OnFocus* event, which fires when the cursor enters the field, causes the field's description to be placed on the browser's status line and changes the field's background color to a pale blue. This provides visual feedback for keyboard-oriented users as to which field the cursor is currently in.

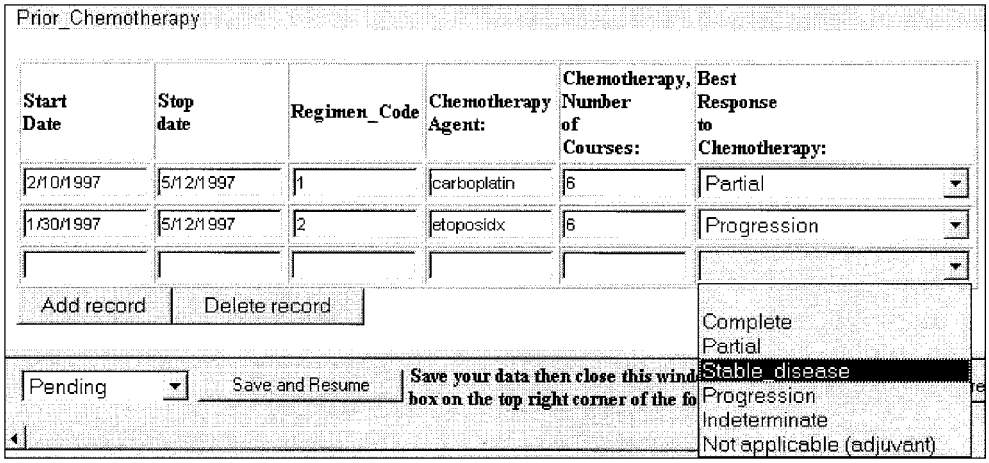

Figure 2 Part of the same data entry form shown in Figure 1 is highlighted to show the insertion of a new, blank row for Chemotherapy (the user has clicked on the "Add Record" button.) Notice how the choices of the "Best Response" pull-down list are copied to the new call.

- Code for the *OnBlur* event, which fires when the cursor leaves a field, whether or not its contents have changed, undoes the effects of *OnFocus*.

Simulation of Subforms

As stated earlier, subforms are needed when there are many-to-one relationships among the data, and we wish to inspect the "many" and "one" records together. WebEAV simulates subforms through HTML tables, using MSIE-specific features. In the MSIE object model, a table's rows and cells are individually accessible programmatically. The number of rows in a table can be changed dynamically and the contents of any new cells that have been created can be set by altering each cell's *innerHTML* property. The contents of the rest of the page are rearranged automatically.

To add a new row to the end of a table, WebEAV accesses the HTML text for each cell in the table's last row. This text is then parsed and appropriately altered, a new row is inserted, and the contents of each corresponding newly created cell is set to the altered text. Figure 2 shows a subform (the form is the same as in Figure 1) with a newly inserted row. Notice that, while the contents of the row are blank, the pull-down menu choices are correctly set.

Suppressing Nonrequired Fields

When the same form is used by multiple departments or for multiple clinical studies, it is desirable to restrict the visible fields on the screen to only those needed by the current user or department. When a user requests a particular form, the WebEAV-generated form template consults the "required fields" metadata on the Web server, inserting a list of "required" fields into the page. The client code then suppresses display of all fields that are not in this list. In MSIE, when one or more objects on a form are programmatically rendered invisible, the form automatically reformats to preserve esthetics.

Advanced Client-side Field Control

Sometimes the forms created by WebEAV need to have computed fields, whose contents are determined by formulas based on other fields in the form. Also, particular values (or ranges of values) that are entered in one field may enable or disable data entry in other fields or may set the contents of other fields to default values. We refer to the latter feature as *skip logic*, because fields are skipped on the basis of the contents of other fields. To complicate matters, the fields involved may be part of a subform, where changes must occur only in the row of data where edits have been made.

In order to parse and evaluate computed formulas or relational expressions that control skip logic, WebEAV relies on a powerful function called EVAL that is built into both VBScript and JavaScript. EVAL takes as input a string representing a syntactically correct (but arbitrarily complex) expression in the language, evaluates this string as though it were code, and returns the result as a value. (LISP was the first language to implement EVAL.)

To tell the browser when to trigger a calculation (or which fields to enable or disable), WebEAV generates program data that specifies dependencies between individual form fields. These data, embedded within the Web page, are consulted by the client-side scripting code. The code is triggered when the user changes the contents of particular fields.

Changes in computed formulas and skip logic are permitted to *cascade*; that is, individual fields can have formulas based on the contents of other fields, some of which are themselves computed. In this way, when the contents of a particular field change, multiple other

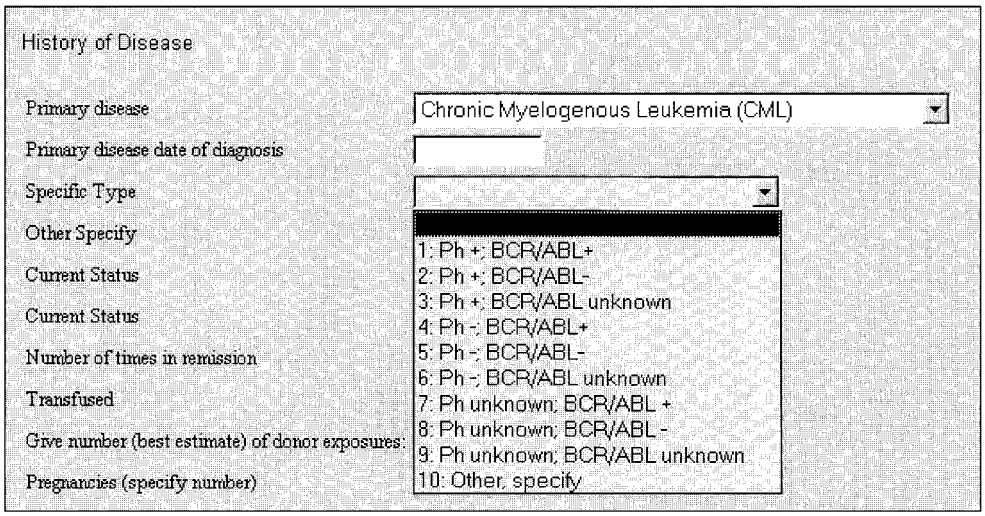

Figure 3 A hierarchic SELECT list in operation. The contents of the "specific type" pull-down box (the *child* list) are determined dynamically by the contents of the "primary disease" pull-down box (the *parent* list). Choices specific to chronic myelogenous leukemia appear here.

fields can have their contents recomputed in "chain-reaction" fashion, in the manner of a spread-sheet.

Hierarchic SELECT Lists

Certain paper questionnaires contain two-part questions, in which responses are specified as lists of items. The item selected in the first part determines the list that is appropriate to the second part. For example, the "Bone Marrow Transplant Recipient" questionnaire of the U.S. National Marrow Donor Program, which is also a form in ACT/DB, has a question entitled "Type of Primary Disease." There are 20 possible responses, including acute myelogenous leukemia, acute lymphoblastic leukemia, Hodgkin's disease, and aplastic anemia. If the user chooses "aplastic anemia," he or she must go to another part of the form where the specific condition is identified—refractory anemia of unknown mechanism, sideroblastic anemia, polycythemia vera, etc. Similarly, if "Hodgkin's disease" is chosen, an item must be selected from a list of Hodgkin's sub-types—lymphocyte depletion, nodular sclerosis, etc.

The paper questionnaire is 15 printed pages long, because it mostly comprises lists of items that may or may not be applicable to a given patient. There is no reason, however, why an electronic form should slavishly follow its paper counterpart. WebEAV implements a feature, *hierarchical SELECT lists*, which can greatly reduce the size of the equivalent electronic form. Here, there are two pull-down boxes, or list boxes, in the form ("SELECT" elements, in Web parlance). The choices offered in the second box change dynamically depending on the choice made in the first. Figure 3 illustrates this feature.

WebEAV implements hierarchic choice sets in a manner similar to that described for computed formulas, by tracking the dependency between "parent" and "child" choice-set boxes on a form. The full list of all child choices for every parent choice is also stored statically in the form's programming data during automatic form generation. The code that changes the contents of the child box is simpler than for computed fields or skip logic, because EVAL does not need to be used.

While the equivalent of hierarchic SELECT lists could be implemented by using skip logic, the former solution is more elegant when applicable, because the resulting form is much smaller.

Status Report

As stated earlier, WebEAV is currently being employed in two production databases: ACT/DB and SENSELAB. ACT/DB currently holds data on 35 studies, spanning the domains of oncology, psychiatry, and cardiology, and has around 3700 attributes grouped into 201 forms. It has 69 users at Yale and collaborating institutions; public access is, understandably, not provided. SENSELAB, whose contents are publicly accessible via http://ycmi.med.yale.edu/senselab/, holds data on around 2,100 objects of various types (olfactory receptors, neurotransmitters, neurons, models, etc.).

The best way to show the status of WebEAV is to provide an example that shows what it takes to use it. We illustrate the user interface of WebEAV for a real study stored in ACT/DB. This study, whose principal investigator is Douglas Bremner of the Yale Department of Psychiatry, concerns post-traumatic stress disorder. The user of this interface is a study designer who does not necessarily know HTML or program-

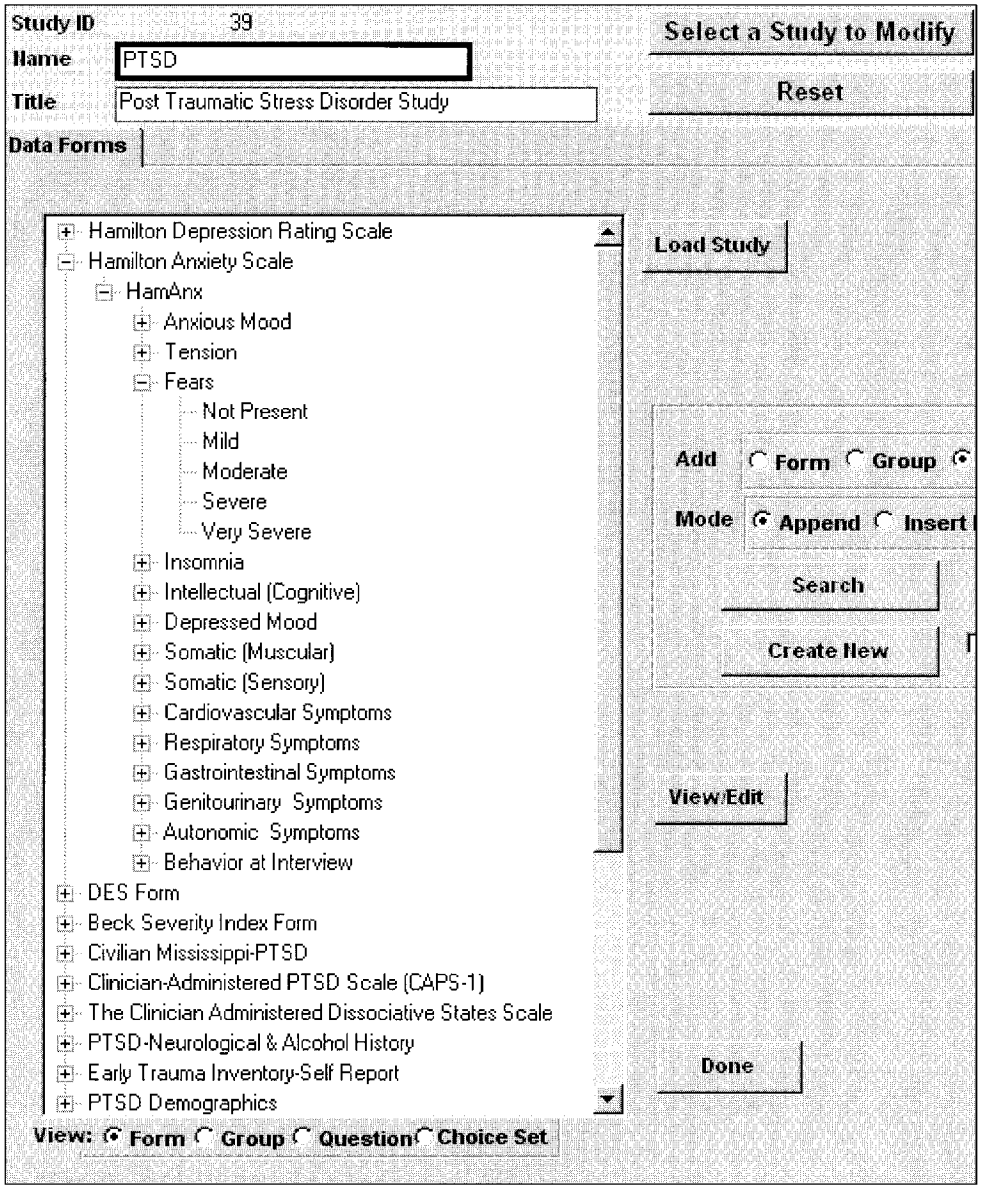

Figure 4 The study designer's interface to WebEAV as implemented in ACT/DB. The screen snapshot shows some of the metadata in a post-traumatic stress disorder study. Data are presented in a hierarchic "outline" view. Double-clicking on any item in the list displays the details of items below it in the hierarchy. The upper third of the figure shows different types of metadata. In order of increasing indenting level, we have a form ("Hamilton Anxiety Scale"), groups of attributes ("HamAnx"), an attribute ("Fears"), and the contents of a choice set for that attribute (Not Present, Mild, Moderate, etc.). The designer can edit the details of a selected item with the "View/Edit" button. A "Generate All Forms" button generates all Web forms for the study from the metadata. An individual form can also be generated, or regenerated, by viewing or editing its details and clicking a "Generate Web form" button.

ming. In WebEAV, generation of forms is metadata-driven. It is therefore critical that the metadata be correct in terms of the designer's intentions and that the designer have a robust and friendly interface for inspecting and editing the metadata to ensure that this is so.

The interface, implemented within the Microsoft Access-based ACT/DB client by the second author, uses the "outlining" metaphor that is well known to users of word-processing and presentation software. That is, the user can get a global perspective of the work at hand or drill down to inspect the details at any level. In this case, the designer can look at all the forms in the study or drill down to details of an individual form, groups of attributes within the form, individual attributes, or (where applicable) the choice set for an attribute. Computational formulas are stored against individual attributes; ACT/DB includes a visual "formula builder" to assist in the creation of formulas. Skip logic information, which applies to the entire form, is accessed from the form level.

Figure 4 shows the interface in operation. When the metadata are edited to the designer's satisfaction, the designer clicks a button (not shown) that generates all Web forms for the study. Alternatively, the designer can generate selected Web forms one at a time; this is typically done when a single form needs regeneration. The generated Web forms are written to a directory on the test Web server. Using the Web browser, the designer then tests a particular form by entering

"dummy" data. If the form's behavior is not what was expected or desired, the designer goes back to the form's metadata, inspects them for logical errors, makes corrections, regenerates the forms, and so forth iteratively. If the designer cannot determine the cause of the error, he or she has been instructed to contact one of the authors, who will determine whether the problem is due to an error in the designer's logic or to a bug in the code library. In our experience, the frequency of logic errors tends to drop gradually as designers get more conversant with the system. Finally, forms that have been fully tested are moved to the production Web server.

In the case of ACT/DB, the form-generation component of WebEAV lies in certain code modules in the ACT/DB Microsoft Access client; that is, we use a traditional client–server application to generate parts of the Web application. The runtime code libraries, whose routines are invoked by code in the generated forms, reside on the Web server. Some of the routines in the latter—e.g., those that perform client-side validation or computed field/skip logic handling—are downloaded to client machines on demand, by being "included" in the Web forms. As stated earlier, the designer is not required to be aware of these components. Developers are at liberty, however, to use some of the runtime routines in their own applications independent of the metadata framework.

Discussion

The WebEAV framework grew out of our earlier work on ACT/DB, partly as a response to frustration with the form-maintenance problems posed by traditional client–server systems. With the traditional solution (Microsoft Access-based data entry) the time of our (few) developers and support personnel was taken up by client-machine support rather than development. Because of this we chose to explore the Web as a delivery medium. The amount of code that needed to be written to perform the most basic tasks (such as validation of dates) caused us to step back and devise a generic approach, which WebEAV embodies. Subsequently, we extended WebEAV for conventional data, by porting SQLGEN,[18] a code library originally intended for traditional client–server development. Although some features of WebEAV are related to biomedical databases (which use the EAV data model to a greater extent than any other domain), much of WebEAV is generalizable.

Several powerful development environments are commercially available for Web programming, such as Microsoft Visual Studio and Apple WebObjects. For access to conventional (non-EAV) data, their use would probably be preferable to that of WebEAV. However, these tools do not attempt to generate interfaces to EAV data, which seem to appear almost exclusively in the biomedical domain and are hardly ever seen in "business" applications. Use of WebEAV, which we intend to distribute as open source, is not necessarily incompatible with such environments: Visual Studio was used to facilitate development and debugging of WebEAV. Developers are free to choose the WebEAV components that they find to be the most valuable.

A question facing the developer is whether a framework similar to WebEAV can be built with alternative software tools. We discuss two alternatives, both of which we briefly considered during WebEAV's conceptual phase, and then abandoned.

Alternative 1: Using Java Applets Within a Page

An advantage of applets for developers is that, because they are downloaded as byte code rather than as source (which is the case with client-side script), intellectual property is protected. However, this is not a concern for us; we are more concerned with permitting customization of machine-generated forms by lead users. A drawback of Java is that while Java's AWT (Abstract Window Toolkit) library has a basic set of objects for user-interface management, it takes more programming effort to use than dynamic HTML, especially when the latter is coupled with cascading style sheets. In particular, enabling client-side validation of data in a generic fashion and permitting dynamic changes in form behavior take considerably more code with Java than with dynamic HTML. While Java applets currently provide a means of richer and more complex user interaction, we did not need such sophisticated interaction for the simple forms-based interfaces that WebEAV generates.

In any case, Java's primacy for use in constructing complex interfaces may soon change. Vector modeling language (VML), whose specifications are described at http://www.w3.org/TR/NOTE-VML, is an application of extended markup language (XML)[21] that is intended to allow the display and editing of vector graphics data over the Web. Vector modeling language, which is currently supported by Microsoft Internet Explorer but not Netscape Navigator, is significantly more powerful than Java for the construction of complex interfaces. Recently available commercial tools such as Visio 2000 let developers work with VML visually. We are currently exploring VML's use in Web display and editing of pedigree data, in the context of another project.

Alternative 2: Using ActiveX Controls on the Client

The latest version of Microsoft Office allows the development of "data access pages," where Web forms can be composed visually from the definitions of tables and columns in the database, and deployed easily. However, these forms support conventional, not EAV, data. Furthermore, their present implementation has a few limitations. Subforms, for example, are read-only and do not allow editing.

More important, the form-generation process inserts ActiveX controls within a page. ActiveX controls are like applets, except that they are compiled binary code. The generated pages are about five times as large as the corresponding page created with straight ASP and embedded script. ActiveX controls, being binary code, will run only on Wintel platforms. (MSIE itself runs on the Macintosh and certain UNIX machines.)

Most users (including the authors) are distrustful of client-side ActiveX controls, with good reason. Unlike Java applets, where the browser runs the the applet's byte code in a "sandbox"[19] to prevent it from performing dangerous operations such as writing to the local disk, the ActiveX model of security is essentially nonexistent, being based on trust in the party that created the control. The "trust" mechanism is based on digital signatures, which, while they are very hard to forge, don't mean much if the party is unknown to the user. An ActiveX control, once accepted by a user, can do anything it chooses, including erasing the hard drive. A more insidious scenario has been constructed,[20] in which one ActiveX control, while not doing any damage itself, disables the local machine's security measures and opens the back door to other, possibly malicious controls.

Future Directions

WebEAV is constantly evolving to meet user needs. The advanced features described earlier, for example, were created in direct response to needs articulated by our collaborators. As Web technology itself evolves, WebEAV will change to take advantage of new features. The HTML standard itself is unlikely to change after the present version (4.0), and new tags that could define the behavior of objects in a page will be added through implementations of XML. Although we have created pilot applications that use XML for the basis of data interchange, at this moment we are unclear how WebEAV, which deals primarily with user interface construction, would benefit from XML.

Individually, each of the features in WebEAV is not exceptionally complex. Put together, however, they benefit the developer significantly. As stated earlier, Web programming is still an arcane art with numerous pitfalls for the unwary developer, and therefore we have described the WebEAV framework in enough detail to permit the reader to understand the algorithmic principles behind our approach.

Availability of Software

The WebEAV framework consists of source code in ASP VBScript and client-side JavaScript or VBScript, plus a Microsoft Access application that accesses metadata schemas to drive the code-generation process. We also include documentation on the routines in the code libraries. We will provide WebEAV at no cost to anyone who makes a written request to us.

Daniel Masys, MD, of the University of California, San Diego, identified the distinction between the global and logical schemas for EAV data. The authors thank Pasquale Cusano of the Connecticut Mental Health Center (CMHC), New Haven, for extensive beta testing and Roy Money of CMHC and David Schoenfeld, of the Department of Biostatistics, Harvard Medical School, for providing valuable suggestions for enhancement. They also thank Douglas Bremner, MD, of the Yale Department of Psychiatry, for permitting the use of the PTSD study forms to illustrate the use of WebEAV.

References ■

1. Huff SM, Berthelsen CL, Pryor TA, Dudley AS. Evaluation of a SQL model of the HELP patient database. Proc 15th Symp Comput Appl Med Care. 1991:386–90.
2. Huff SM, Haug DJ, Stevens LE, Dupont CC, Pryor TA. HELP the next generation: a new client–server architecture. Proc 18th Symp Comput Appl Med Care. 1994:271–5.
3. 3M Health Information Systems. 3M Clinical Data Repository. Murray, Utah: 3M Corporation, 1998.
4. Friedman C, Hripcsak G, Johnson S, Cimino J, Clayton P. A generalized relational schema for an integrated clinical patient database. Proc 14th Symp Comput Appl Med Care. 1990:335–9.
5. Johnson S, Cimino J, Friedman C, Hripcsak G, Clayton P. Using metadata to integrate medical knowledge in a clinical information system. Proc 14th Symp Comput Appl Med Care. 1990:340–4.
6. Nadkarni P, Brandt C. Data extraction and ad floc query of an entity-attribute-value database. J Am Med Inform Assoc. 1998;5:511–27.
7. Nadkarni PM, Brandt C, Frawley S, et al. Managing attribute-value clinical trials data using the ACT/DB client–server database system. J Am Med Inform Assoc. 1998;5: 139–51.
8. National Cancer Institute. The Cancer Genetics Network. Details available at: http://www-dccps.ims.nci.nih.gov/CGN/. Accessed May 17, 2000.
9. Shepherd GM, Healy MD, Singer MS, et al. SenseLab: a project in multidisciplinary, multilevel sensory integration.

In: Koslow SH, Huerta MF (eds). Neuroinformatics: An Overview of the Human Brain Project. Mahwah, NJ: Lawrence Erlbaum, 1997:21–56.
10. Shepherd G, Mirsky JS, Healy MD, et al. The Human Brain Project: Neuroinformatics tools for integrating, searching and modeling multidisciplinary neuroscience data. Trends Neurosci. 1998;21(11):460–8.
11. Nadkarni PM, Marenco L, Chen R, Skoufos E, Shepherd G, Miller P. Organization of heterogeneous scientific data using the EAV/CR representation. J Am Med Inform Assoc. 1999; 6:478–93.
12. World Wide Web Consortium. Cascading Style Sheets. Available at: http://www.w3.org/style/css. Accessed May 17, 2000.
13. Dwight J, Erwin M (eds). Special Edition: Using CGI. Indianapolis, Ind: Que Corporation, 1996.
14. Darwin IF. Checking C Programs with Lint. Sebastopol, Calif: O'Reilly Associates, 1988.
15. Sun Microsystems. Information about and technical overview of Java server pages is available at: http://java.sun.com/products/jsp. Accessed May 17, 2000.
16. Goodman D. Dynamic HTML: The Definitive Reference. Sebastopol, Calif: O'Reilly Associates, 1998.
17. Graham IS. HTML 4.0 Sourcebook. New York: Wiley Computer Publishing, 1997.
18. Nadkarni PM, Cheung KH. SQLGEN: an environment for rapid client–server database application development. Comput Biomed Res. 1995;28(12):479–99.
19. Oaks S. Java Security. Sebastopol, Calif: O'Reilly Associates, 1998.
20. Goncalves M. Firewalls Complete. New York: McGraw-Hill, 1998.
21. Boumphrey F, Di Renzo O, Duckett J, et al. XML Applications. London, UK: WROX Press, 1998.