Representing Procedural Logic in XML

Albert D. Bethke
RTI International, Research Triangle Park, NC, 27709, USA
adb@rti.org

***Abstract*—Extensible Markup Language (XML) is a powerful tool used for describing structured documents and exchanging standardized data files over the Internet. This article describes how using XML in an unconventional way greatly improves the usability and effectiveness of an authoring system for generating computer-assisted interviewing (CAI) applications. In addition to specifying the content, structure, and format of a questionnaire, XML tags are used to specify the procedural elements (Boolean expressions and simple computations) that represent the dynamic aspects of a CAI questionnaire. These procedural elements are represented with the creation of a set of XML tags that embody a simple functional programming language.**

***Index Terms*—extensible markup language, computer-assisted interviewing, computer-assisted self-interviewing, functional programming**

I. Introduction

Extensible Markup Language (XML) [1] is designed to represent structured documents and data sets [2]. XML is an open standard that allows organizations to create special-purpose markup languages by defining a set of XML tags. There are no predefined or "standard" meanings for XML tags. Each special-purpose XML-based markup language defines the structure and meanings of its tags. By design, all of these languages can be parsed by a single standard XML reader that is small, fast, and simple to implement.

XML is used to exchange data in standard "streaming" format over the Internet and elsewhere. Many organizations have adopted XML data exchange standards for this purpose [3]–[6]. These organizations include the U.S. Congress, bioinformatics associations, retail product distributors, and others. XML is also used to describe and implement Web services [7]—which is really an example of using XML for standardized data exchange.

Surveys and interviews are used to collect various kinds of information from a wide range of target populations. Computer-assisted interviewing (CAI) [8] allows for more flexible and effective interviews than paper-and-pencil questionnaires.

This journal article is based on the conference paper, "Using XML as a Questionnaire Specification Language," which appeared in the *Proceedings of IEEE SoutheastCon 2007*, Richmond, VA, March 2007, © 2007 IEEE.

RTI International conducts a large number of CAI surveys on behalf of government and commercial clients. RTI has recently designed and implemented an authoring system for CAI surveys that uses XML as the questionnaire specification language [9]. This system is called the *Simple Survey System* (SSS).

This article describes how using XML in an unconventional way greatly improves the usability and effectiveness of the SSS. In addition to specifying the content, structure, and format of the questionnaire, XML tags were used to specify the procedural elements (Boolean expressions and simple computations) that represent the dynamic aspects of a CAI questionnaire. Representing all aspects of the questionnaire in a single XML specification file provides several advantages:

- It eliminates the need to reprogram the survey engine for each questionnaire.
- It allows for the easy development of authoring tools that can be used by nonprogrammers (survey designers or research assistants) to fully develop complete CAI systems.
- It makes it easy to deploy and maintain the CAI application in the field.

The rest of the article frames the problems to be solved, presents the approach used to solve them, and gives details and examples of the XML representation that was developed.

II. Computer-Assisted Interviewing

The federal government sponsors a large number of surveys every year on topics ranging from drug use to personal driving habits to the benefits of student loans. In addition, many companies conduct marketing surveys, customer satisfaction surveys, and other types of surveys. Many, if not most, of these surveys are done with CAI technology.

A. Advantages of CAI

CAI surveys offer several advantages over paper-and-pencil surveys, including the following:

- Data are validated during collection.
- Data keying errors are eliminated.
- Complex, sophisticated routing is practical.
- Question wording can be dynamically customized using fills to make some items easier to understand or more specific.

With CAI, the interviewer must provide a correct response before he or she can proceed to the next item.

© 2008 ACADEMY PUBLISHER

For simple multiple-choice items, this means that the interviewer must select exactly one of the options that appear on screen. With a paper-and-pencil questionnaire, the interviewer may mark more than one response, may fail to mark any responses, or may mark the form in such a manner that it is not clear which response option was selected. For other types of CAI questions, the response will be validated by verifying that it is within an acceptable range of values or that it is consistent with previous answers. If the value is not acceptable, the interviewer is immediately informed of the problem and prompted to enter a new value.

With paper-and-pencil surveys, it is necessary to key the data into a data file or database after the questionnaire has been completed. Although it is possible to achieve very low error rates for this data entry task, it is time consuming and expensive, and a few errors always slip through. CAI data are directly recorded in a database or data file, thereby eliminating the time and expense, as well as the errors, associated with keying the data from paper forms.

Paper questionnaires often include routing instructions to the interviewer so that certain questions (or whole sections) are skipped if they are not appropriate for a specific respondent. Such instructions generally appear immediately following a question or immediately before a question or section. A routing instruction might be something like, *If the answer to Q6 is NO, skip to Q22*.

Experience has shown that even well-trained professional interviewers frequently fail to correctly follow routing instructions. This failure results in missing data for those items that were mistakenly skipped. It may also confuse the respondent with inappropriate questions and lead to incorrect answers later because the respondent is annoyed or confused. CAI surveys, on the other hand, can implement complex, sophisticated routing logic that is followed without fail. (Achieving this standard requires careful testing and debugging of the application.)

One more advantage of CAI surveys is that they allow adjustment of question wording to improve understanding and make items fit a specific respondent. For example, after asking a respondent about his job history, a follow-up question might ask about his experiences with a particular employer, like so: *When you worked at Bell Labs, were you ever required to work more than 40 hours in one week?* In this question, *Bell Labs* has been taken from a previous response and used as fill text.

B. Computer-Assisted Personal Interviewing

Computer-assisted personal interviewing (CAPI) refers to in-person interviews conducted by a professional interviewer using a laptop, tablet, or handheld computer. A trained interviewer starts the CAPI application running on the computer and then reads the questions to the respondent and records the responses. The CAPI application advances to the next item as each response is entered.

C. Computer-Assisted Self-Interviewing

Computer-assisted self-interviewing (CASI) typically is done with a laptop computer. A "facilitator" starts the interview, shows the respondent how to operate the CASI application, and then moves away and lets the respondent directly enter his or her responses in relative privacy.

Research has shown that CASI administration produces more honest, more complete responses to questions about sensitive topics, such as illegal drug use, risky or unusual sexual behavior, and similar topics [10], [11]. So, for sensitive topics, it is customary to use CASI, which allows the respondent to directly enter his or her answers into a computer rather than share potentially embarrassing information with an interviewer.

Audio-CASI (ACASI) is useful for respondents who may have difficulty reading questions on the computer screen. With ACASI, the computer "reads" each question to the respondent. This administration can be accomplished with audio files that are recorded in advance for each question and all the response options. It can also be accomplished with a text-to-speech application, but this practice is much less common.

D. Authoring Systems and Self-Interviews

RTI International has completed hundreds of projects that used CAI. Several commercial authoring systems are available that simplify the process of developing and conducting CAI surveys. Two of these commercial systems are frequently used by RTI for telephone surveys and face-to-face personal interviews: CASES [12] and Blaise [13]. Both of these packages offer a special-purpose questionnaire programming language, together with tools for managing the data that are collected and for tracking the status of interviews.

As already described, for self-interviews the CAI application user is the respondent, not a trained professional interviewer; therefore, the application user has no experience with the interview application prior to starting the interview. It is not practical to train the respondent to use a complicated interface to operate the application; the interface must be as simple and as intuitive as possible.

Commercially available CAI authoring systems are usually oriented toward telephone and face-to-face personal interviews conducted by professional interviewers. They provide a rich interface for controlling the application to allow the interviewer to back up or skip forward quickly or to break off the interview and schedule a follow-up session and so on. This kind of system provides desirable flexibility and power for the interviewer but requires significant training before the interviewer is comfortable with the application. Because commercial CAI authoring systems do not provide the simple intuitive interface needed for self-interviews, RTI has developed an authoring system for this purpose. The next section describes this authoring system.

III. The Simple Survey System

A. Previous Work

Although the SSS uses XML now, it did not start that way. A prior system was developed before XML was in common use; looking at the implementation history and

© 2008 ACADEMY PUBLISHER

reasons for changing to XML is informative. Questionnaires were represented as structured text files made up of questions grouped into sections. Sections and questions were delimited in the file by specially formatted lines that effectively acted as start and end tags with attributes, although they were not in XML syntax. We will refer to this system as the *original (authoring) system.*

CAI questionnaires are not static documents. Different respondents will see different questions. For example, in a questionnaire about health and lifestyles, smokers may be asked questions that nonsmokers will not see—for example, *Have you ever tried to quit smoking?* Question text is often customized to better fit a specific respondent by filling in phrases based on previous responses. Routing decisions and selection of fill text are often based on values computed from several responses. So to fully represent a CAI questionnaire, we need to specify how to compute these values and the routing decision logic and fill variables that control what is presented on screen.

In the original authoring system, routing logic, fills, computed variables, and all procedural elements of the questionnaire were coded in the survey engine. We modified the survey engine for each new questionnaire, starting each time from a standard base version of the code.

In the original system, the survey engine read the questionnaire specification file at the start of the interview and created arrays of section and question objects in memory to capture the structure, content, and format of the questionnaire. Before and after the survey engine presented each question, it executed routing logic code and any code related to fills and computed variables that may have been associated with the next question. (This step might have changed the text to be displayed or might have caused the particular question to be skipped.) It is a relatively straightforward process for a C++ programmer to implement this code from the specifications typically provided by a survey designer.

We recognized that it was desirable to include the procedural elements in the questionnaire specification file, but we could find no reasonable way to do so. Commercial CAI authoring systems provide special programming languages for this purpose. For example, CASES uses a FORTRAN-like syntax for its programming language, and Blaise provides an extension to PASCAL. We considered adding such a language to the original authoring system. However, the effort to design and implement such a language would far exceed the total effort for developing the original system. So we managed by reprogramming the survey engine for each project.

B. Why Use XML?

Recently, we redesigned and reimplemented the authoring system, the SSS. Because a questionnaire is a structured document and because the original system used a specially formatted text file to represent a questionnaire, we considered using XML for the questionnaire specification language. We also considered storing the questionnaire specifications in a database. In fact, we first considered using a database, but rejected the idea after comparing it to using XML.

Using a database to store the questionnaire specifications is an attractive approach because the same database can be used to store the response data. The database schema for representing questionnaires is not difficult to develop or implement. Using a database might be the best choice for Web surveys or telephone surveys that can use a single, centralized database server. But for CASI and ACASI surveys done in the field with many laptop computers, the XML approach offers significant benefits:

- It is very easy to set up the field laptop computers because there is no database management system (DBMS) to install or configure.
- Maintaining the field computers is also easy because there are never any problems related to the DBMS (since there is no DBMS).

As for storing the response data, the SSS stores it in XML data files, so no DBMS is needed for this purpose. As a result, installing an SSS survey application consists of simply copying a few files to the hard drive of the laptop.

C. Version 1 of the SSS

The first version of the SSS was very similar in overall architecture to the original system. The questionnaire specification language was more carefully designed, and it was implemented in XML. As a result, the questionnaire specification language parser was simpler, more robust, and more efficient. In addition, extending the questionnaire specification language as new types of questions were developed or new formatting options were requested was simple and straightforward because XML is designed to have this flexibility.

One of the design goals for the first version of the SSS was to simplify and standardize the way that the routing logic and computed variables are handled. The SSS improved on the previous system by consolidating all the procedural elements into a single module with a very simple structure. It was an easy job for a programmer to add the necessary code for routing logic and computed variables to this module as required for each questionnaire.

This version of the SSS was used quite successfully for a large federally funded survey project. Four versions of a lengthy questionnaire were developed for males and females in English and Spanish. The questionnaire was rather substantially revised several times. The SSS proved to be cost-effective and easy to use—a clear improvement over the original authoring system.

D. Reducing the Programming Effort

The next project that used the SSS was even more demanding in terms of producing a large number (35) of lengthy questionnaires that were revised many times before the final version was agreed upon.

One of my colleagues developed what we call the *Automated Builder* [9] to enable a research assistant to prepare the questionnaire specification files. The research assistant was not a programmer, and she did not

© 2008 ACADEMY PUBLISHER

understand XML. The survey designer provided questionnaire specifications as Word documents that could be printed to produce a paper questionnaire. Routing instructions, fills, and computed variables were described in a loosely defined pseudo-code within the document.

The Automated Builder presents a graphical interface that the research assistant uses to enter the specifications for the questionnaire from the Word document. After the questionnaire is fully specified in this way, the Automated Builder produces the XML questionnaire specification file for the SSS with a single mouse click.

At this stage of development, we were using version 1 of the SSS, so the routing logic and computed variables were not entered by the research assistant and were not included in the questionnaire specification file. The procedural elements of the questionnaire were programmed into the survey engine by C++ programmers after the questionnaire specification file was generated.

Nonetheless, by using the Automated Builder, the project was able to reduce costs, reduce demand for the limited pool of programmers, and reduce overall development time.

E. Completing the Questionnaire Specification Language

The need to program the routing logic and computed variables for 35 different questionnaires into the survey engine motivated us to reconsider how the routing logic could be represented in the XML specifications. Routing logic is usually represented as Boolean expressions, with the variables being previous responses. A simple example would be, *If the answer to Q6 is NO, skip to Q22*.

Functional programming languages such as LISP and Prolog facilitate the specification of complex computations using only a very simple syntax and semantics [14]. So the key to representing routing logic in XML is to use a functional programming approach and to use XML tags to represent function invocations. The examples in the next section will be easily understood by anyone who has experience with LISP (or other functional) programming.

The common way that routing logic is presented is to use forward references and to skip over questions or whole sections. However, an alternative to skip logic that is sometimes used is *gates*—expressions that specify when a question or section should be asked rather than when it should be skipped.

Adding a gate tag to be used as the first element of a question or section is a simple way to implement routing logic in XML. When survey designers use skip logic, we just reverse the logic and create a gate instead.

Version 2 of the SSS allows the full specification of the questionnaire, including dynamic behavior, in the XML specification file. XML is not ideally suited as a programming language—the code is not exactly elegant and concise—but putting the code into XML means that it is not necessary to create another programming language and implement a parser and interpreter for that language.

The implementation history of the SSS is summarized in Table 1. The current Automated Builder does not handle routing logic or computed variables. The procedural elements of the specification are manually added to the specification file by programmers after the initial file is prepared by a research assistant using the Automated Builder. We plan to make a future version of the Automated Builder to allow a research assistant to easily specify routing logic and computations and include these in the XML file it generates, thereby eliminating the need for manual programming related to the questionnaires.

The next section will present the XML schema I designed for representing a simple programming language using XML tags. Notice how this use differs from the normal uses of XML to represent the structure and content (and possibly the format) of structured documents or data sets. Consider that XML is part of the popular AJAX [15] Web programming paradigm. In the AJAX paradigm, JavaScript, Visual Basic, or server-side programming languages (like Java or Perl) are used to handle the procedural elements of the Web pages—XML is not used for this purpose.

IV. FUNCTIONAL PROGRAMMING IN XML

Fig. 1 gives a portion of the XML schema definition (XSD) for the SSS questionnaire specification language. All the elements related to functional programming are shown in Fig. 1.

TABLE I.
HISTORICAL DEVELOPMENT OF THE SSS

Version	Features
Original Authoring System	• Text file specifies structure and content, but not procedural aspects of questionnaire. • Specification file is coded manually by programmer. • Survey engine is reprogrammed for each questionnaire to handle routing logic, fills and computed variables.
SSS Version 1	• XML specifies structure and content, but not procedural elements of questionnaire. • Automated Builder produces specification file. • Survey engine is reprogrammed for each questionnaire to handle routing logic, fills, and computed variables.
SSS Version 2	• Complete XML specifications include routing logic and computed variables. • Automated Builder produces specification file, but programmer must manually add routing logic, fills, and computed variables to specification file. • Same survey engine used for all questionnaires – no reprogramming needed.
SSS Future Version	• Complete XML specifications include routing logic and computed variables • Automated Builder produces complete specification file, including routing logic, fills and computed variables • Same survey engine used for all questionnaires—no reprogramming needed

© 2008 ACADEMY PUBLISHER

Working through the definitions in Fig. 1, we can see that a gate is a Boolean (integer) expression that determines whether the associated section or item will be presented. The XSD specifies that exactly one expression will appear within the gate element. (It does not specify that the expression must have an integer value.)

An expression is either a constant, variable, or function invocation. Constants are specified using <integer> or <string> tags, with a string representing the constant between the start and end tags. Variables are simply specified using an empty variable element with the name attribute uniquely identifying the variable. (The XSD does not specify that the variable name must be unique.) Functions are also identified by the name attribute, and the subelements are expressions to be evaluated as the arguments for the function.

As the examples are presented in the next few sections, it may be helpful to refer to this XSD.

A. Routing Logic / Gate Example

Fig. 2 shows the specifications for an unrealistically simple survey about ice cream as they might be provided by a survey designer. This survey consists of an introductory informational item, two multiple-choice questions, and a final informational item. The survey is not divided into sections. The routing logic is embedded as a "comment" beside the *No* option for Q1.

Fig. 3 shows how this survey would be represented in the SSS query specification language using XML tags. It was necessary to create a section (arbitrarily named "A") to contain the questions, because the SSS requires this structure. Notice that the skip logic from the survey designer's specification has been reversed to become the gate expression for Q2. The gate expression is the result of invoking the function named "NE" (not equal) with two arguments. The first argument is the variable named "Q1." The second argument is an integer constant with the value 2—the same value as the *No* option for Q1.

As it reads and parses the questionnaire specification file, the survey engine automatically creates the Q1 variable to represent the response value for the multiple-choice question named "Q1." As part of the normal processing cycle, the SSS survey engine will save the value of the selected option in the variable Q1 immediately after the user makes a selection for question Q1. This value will be available thereafter. (All variables associated with multiple-choice questions are initialized to a special value that represents "missing" or "not answered.")

I decided to use a general "function" tag with the "name" attribute to specify a particular function rather than to create distinct tags for each function. This option was chosen to reduce the number of XML elements, which simplified the development of the XML parser. The disadvantage of this approach is that the number and type of arguments required by a specific function cannot

```
<xsd:element name="gate">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="Expression" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:group id="Expression">
  <xsd:choice>
    <xsd:group ref="Constant" />
    <xsd:element ref="variable" />
    <xsd:element ref="function" />
  </xds:choice>
</xsd:group>

<xsd:group id="Constant">
  <xsd:choice>
    <xsd:element name="integer" type="xsd:integer" />
    <xsd:element name="string" type="xsd:string" />
  </xsd:choice>
</xsd:group>

<xsd:element name="variable">
  <xsd:complexType>
    <xsd:attribute name="name" type="xsd:string" use="required" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="function">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="Expression" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required" />
  </xsd:complexType>
</xsd:element>
```

Figure 1. XML schema definition for procedural elements of questionnaires.

© 2008 ACADEMY PUBLISHER

```
ICE CREAM SURVEY
INTRO:  This is a survey about ice cream.
Q1:     Do you like ice cream?
             1. Yes
             2. No [skip to END]
Q2:     What is your favorite flavor?
             1. Vanilla
             2. Chocolate
             3. Some other flavor
END:    Thank you for taking the survey.
```

Figure 2. Questionnaire specification from survey designer.

be validated when the questionnaire specification file is read. Argument validation must wait until the function is actually invoked. This limitation has not proven to be a problem in practice; nonetheless, it may be changed in the future because having distinct tags for each function allows one to detect argument-type errors simply by reading the specification file rather than by exhaustively testing every path through the questionnaire.

As specified in the XML schema definition for the questionnaire specification language (see Fig. 1), expressions are either constants, variables, or function invocations. Complex expressions are built up as nested function calls. Expressions are evaluated recursively, from the inside out. The recursion terminates when variables or constants are evaluated. The survey engine has classes for constants, variables, and functions; each of these classes has its own function for parsing the associated XML element and another function for interpreting and evaluating that kind of expression. This design results in a clean and simple implementation that yields excellent performance.

An example of a moderately complex gate expression is shown in Fig. 4. This gate expression illustrates the way function calls are nested to build up more complex expressions. In English, the expression is "(Q1 equals YES) or ((Q2 is less than 3) and (Q6 is not equal to 7))."

Currently, the SSS supports only integer and string variables. In the future we will incorporate date-time variables and currency variables into the SSS. Boolean expressions are really integer expressions with the C language convention that zero acts like false and nonzero values act like true. The SSS includes functions for basic integer arithmetic, setting and concatenating strings, relational comparison of integers or strings, and Boolean operators. In addition, special functions provide procedural (sequential) control constructs such as *if-then-else* and *while*. Although not the most elegant and compact representation for such expressions, XML serves well for this purpose, and we avoid needing to design and implement a separate "computational" language with its own special syntax.

B. Computation Example

As with gate expressions, computation is specified with a functional representation. Assigning values to variables is done with the "SET" function. The SET function takes two arguments: a variable and an expression of the same type as the variable. As one would expect, SET changes the value of the variable to be the value of the expression.

Fig. 5 is an example of using IF, THEN, and ELSE functions to set a fill variable to either "he" or "she," depending on the gender of the respondent (as indicated in the response to a previous question). The IF function takes three arguments: a Boolean (integer) expression, an invocation of the THEN function, and an invocation of the ELSE function. As one might guess, the Boolean expression is evaluated, and, if it is true (nonzero), the THEN function is invoked and the ELSE function is not

```
<questionnaire name="IceCream" title="Ice Cream Survey">
	<section name="A">
		<textItem name="INTRO">
			<text>This is a survey about ice cream.</text>
		</textItem>
		<multipleChoice name="Q1">
			<text>Do you like ice cream?</text>
			<choice value="1">Yes</choice>
			<choice value="2">No</choice>
		</multipleChoice>
		<multipleChoice name="Q2" type="MULTIPLE_CHOICE">
			<gate>
				<function name="NE">
					<variable name="Q1">
					<integer>2</integer>
				</function>
			</gate>
			<text>What is your favorite flavor?</text>
			<choice value="1">Vanilla</choice>
			<choice value="2">Chocolate</choice>
			<choice value="3">Some other flavor</choice>
		</multipleChoice>
		<textItem name="END">
			<text>Thank you for taking the survey.</text>
		</textItem>
	</section>
</questionnaire>
```

Figure 3. XML representation of ice cream survey.

© 2008 ACADEMY PUBLISHER

```
<gate>
      <function name="OR">
            <function name="EQ">
                  <variable name="Q1">
                  <variable name="YES">
            </function>
            <function name="AND">
                  <function name="LT">
                        <variable name="Q2">
                        <integer>3</integer>
                  </function>
                  <function name="NE">
                        <variable name="Q6">
                        <integer>7</integer>
                  </function>
            </function>
      </function>
</gate>
```

Figure 4. A moderately complex gate expression.

invoked. On the other hand, if the Boolean expression is false (zero), the ELSE function is invoked and the THEN function is not invoked. So the survey engine will not evaluate all the arguments to the IF function, which is precisely the desired effect.

Programming like this, using XML tags, is not difficult. The functional approach and the matched start and end tags provide a simple, consistent, yet very expressive structure for representing calculations and expressions. However, the resulting "code" is rather lengthy compared to equivalent code in a traditional programming language. For example, the C++ code equivalent to Fig. 5 is shown in Fig. 6: it is much more compact than the XML version.

We need to do only relatively simple programming within the questionnaire specification file, and the advantages of having this code incorporated into the same specification file in the same way as the content and structure are specified (as XML elements) far outweigh the disadvantage of the code's being somewhat lengthy. In fact, the same sort of tradeoffs can be seen when XML is used as a standard data exchange method. Tab-delimited text files and comma-separated values files are much more compact than equivalent XML data files. Nonetheless, because XML provides a simple, powerful, flexible representation that can capture arbitrarily complex data structures and allows for easy future extensions, it has become the standard for data exchange on the Internet.

V. CONCLUSION

An ideal CAI authoring system would enable a survey designer to develop a CAI application without assistance from a computer programmer and without understanding anything about XML or other technical computer topics. The SSS does not quite achieve this ideal, but it greatly reduces the need for custom programming and allows for the rapid development and modification of CAI questionnaires.

Representing a simple functional programming language in XML allows the complete questionnaire specification to be presented in a simple, uniform manner in a single file. Because the representation is XML-based, developing the robust parser and interpreter for the specification language incorporated into the SSS survey engine was relatively easy. Finally, this solution facilitated the development of the Automated Builder, which allows the rapid development of CAI questionnaires with minimal assistance from professional programmers.

```
<function name="IF">
    <function name="EQ">
         <variable name="gender" />
         <variable name="MALE" />
    </function>
    <function name="THEN">
         <function name="SET">
            <variable name="pronounFill" />
            <constant type="TEXT" value="he" />
         </function>
    </function>
    <function name="ELSE">
         <function name="SET">
            <variable name="pronounFill" />
            <constant type="TEXT" value="she" />
         </function>
    </function>
</function>
```

Figure 5. XML specification of an if-then-else computation.

© 2008 ACADEMY PUBLISHER

```
if (gender == MALE)
       pronounFill = "he";
else
       pronounFill = "she";
```

Figure 6. C++ code for if-then-else computation.

ACKNOWLEDGEMENT

I would like to thank Rita Thissen, Susanna Cantor, Craig Hollingsworth, and Jenny Foerst for reviewing early drafts and providing helpful suggestions during the preparation of this article.

REFERENCES

[1] W3C, "Extensible Markup Language (XML)," http://www.w3.org/XML/

[2] J. Bosak, "Media-independent publishing: Four myths about XML," *IEEE Computer*, vol. 31, pp. 120–122, October 1998.

[3] J. Guo et al., "CLAIM (CLinical Accounting InforMation)—An XML-based data exchange standard for connecting electronic medical record systems to patient accounting systems," *J. Med. Syst.*, vol. 29, pp. 413–423, August 2005.

[4] U.S. House of Representatives, "Drafting legislation using XML at the U.S. House of Representatives," http://xml.house.gov/drafting.htm

[5] European Medicines Agency, "Data exchange standard," http://pim.emea.europa.eu/des/index.html

[6] HUPO Proteomics Standards Initiative, "Molecular interaction XML format documentation version 1.0," http://psidev.sourceforge.net/mi/xml/doc/user/

[7] D. Booth et al., "Web services architecture," February 2004, http://www.w3.org/TR/ws-arch.

[8] M. Couper et al., *Computer Assisted Survey Information Collection*. New York: Wiley, 1998.

[9] A. D. Bethke, M. Daniels, and D. Fleischmann, "Simple Survey System for computer-assisted interviews," *Proc. of 11th World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando, FL, July 2007.

[10] P. C. Cooley et al., "Using touch screen audio-CASI to obtain data on sensitive topics," *Comput. Human Behav.*, vol. 17, pp. 285–293, May 2001.

[11] K. G. Ghanem, H. E. Hutton, J. M. Zenilman, R. Zimba, and E. J. Erbelding, "Audio computer assisted self interview and face to face interview modes in assessing response bias among STD clinic patients," *Sex. Transm. Infect.,* Vol. 81, pp. 421–425, 2005.

[12] Computer-Assisted Survey Methods Program, University of California at Berkeley, "Computer-Assisted Survey Execution System," http://socrates.berkeley.edu:7504/

[13] Statistics Netherland, "Blaise Software," http://www.cbs.nl/en-GB/menu/informatie/onderzoekers/blaise-software/default.htm

[14] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, pp. 359–411, September 1989.

[15] Wikipedia, "Ajax (programming)", http://en.wikipedia.org/wiki/Ajax_(programming)

Albert D. Bethke received his Ph.D. degree in Computer and Communication Sciences from the University of Michigan in 1980. He works as a Senior Research Programmer/Analyst for RTI International in Research Triangle Park, NC. Dr. Bethke is a member of Sigma Xi, the scientific research society.

© 2008 ACADEMY PUBLISHER