Improving Teaching and Learning through Automated Short-Answer Marking

Raheel Siddiqi, Christopher J. Harrison, and Rosheena Siddiqi

Abstract—Automated short-answer marking cannot "guarantee" 100 percent agreement between the marks generated by a software system and marks produced separately by a human. This problem has prevented automated marking systems from being used in high-stake short-answer marking. This paper describes how an automated short-answer marking system, called IndusMarker, can be effectively used to improve teaching and learning in an Object-Oriented Programming course taught at a university. The system is designed for factual answers where there is a clear criterion for answers being right or wrong. The system is based on *structure matching,* i.e., matching a prespecified structure, developed via a purpose-built *structure editor*, with the content of the student's answer text.

Index Terms—Automatic assessment tools, self-assessment, classroom feedback systems, computer science education.

1 Introduction

Advances in computer technology and the widespread availability of ever more sophisticated computer-based systems is gradually changing the way teaching and assessment is undertaken in computer science and engineering departments [1]. Educational institutions must explore and effectively utilize opportunities provided by such technologies to enhance the educational experience of their students. Automated short-answer marking is one of the technologies that may be usefully explored, and in recent years a number of attempts have been made to automate short-answer marking [2], [3], [4], [5]. The most important problem addressed by all of these systems is the accuracy with which automated marking can be performed. It is quite reasonable to suggest that it is very difficult to produce an automated system that can guarantee 100 percent agreement with a human marker. Given this limitation, can an automated short-answer marking system have any practical application? The purpose of this paper is to describe an automated short-answer marking system that can be utilized to improve teaching and learning in an Object-Oriented Programming (OOP) course. The OOP course is taught at Bahria University (BU) in Pakistan. The system has been used by educators to monitor the overall performance of students during term-time and to provide feedback to students about their performance.

2 The Object-Oriented Programming Course

Since the release of the Java language more than 10 years ago, OOP has become widely used to introduce programming skills to computer science students [6]. The OOP course taught at BU (a federally chartered university in Pakistan that is accredited by both Higher Education Commission, Pakistan, and Pakistan Engineering Council [8], [9], [10]) aims to introduce programming and object-oriented concepts to undergraduate students using C++ [7]. The course involves three hours of lectures and three hours of laboratory work in each week of its 16-week duration. The goal of the course is to teach students how to develop object-oriented programs, rather than teach details of the C++ language, i.e., the intention is to use the C++ language merely as a tool for mastering object-oriented programming. The course is part of many degree programs at BU including Bachelor of Engineering (Electrical), Bachelor of Software Engineering, Bachelor of Computer Engineering, and Bachelor of Science (Computer Science). BU has two campuses: one in Karachi and the other one in Islamabad. A typical cohort of students taking the OOP course at a BU campus ranges from 200 to 250 students. English is the medium of instruction and computer-based classrooms (i.e., classrooms where each student is provided with a computer) are available. All computers in these computer-based classrooms have a fast internet connection.

3 IndusMarker and the Question Answer Language (QAL)

Questions are an essential component of effective instruction [11]. If questions are effectively delivered, they facilitate student learning and thinking and provide the opportunity for academics to assess how well students are mastering course content [12]. Questions may be categorized as short answer, multiple choice, essay, etc. The two most commonly used categories are multiple-choice and short-answer questions [13].

In the case of the authors' research, "short-answers" implies free-text entry, requiring answers that have to be constructed rather than selected, ranging from phrases to three to four sentences. Moreover, the authors' research is about objective questions rather than subjective questions. The criterion of right and wrong for an answer must be

- *Raheel Siddiqi and C.J. Harrison are with the School of Computer Science, The University of Manchester, Kilburn Building, Oxford Road, Manchester M13 9PL, UK. E-mail: Raheel.Sidddiqi@postgrad.manchester.ac.uk, christopher.j.harrison@manchester.ac.uk.*
- *Rosheena Siddiqi is with the Department of Computer Science and Engineering, Bahria University (Karachi Campus) 13, Stadium Road, Karachi 75260, Pakistan. E-mail: siddiqi.rosheena@gmail.com.*

Manuscript received 30 Oct. 2009; revised 8 Dec. 2009; accepted 29 Jan. 2010; published online 9 Feb. 2010.
For information on obtaining reprints of this article, please send e-mail to: lt@computer.org, and reference IEEECS Log Number TLT-2009-10-0148.
Digital Object Identifier no. 10.1109/TLT.2010.4.

clear. An example of a short-answer objective question is: "What is the main difference between structures and arrays?" The correct answer is: "Arrays can only hold multiple data items of the same type, but structures can hold multiple data items of different data types." Correct student responses are expected to be paraphrases of this concept, and therefore, the primary task of the automated marking system is to recognize which answers are paraphrases of the correct concept and which are not.

Another term used in the paper is "practice tests." Practice tests are low-stake tests taken during term-time. Marks obtained in these tests are not counted toward the final grade of students. The main aim of practice tests is to promote students' learning and regular revision of course contents during term-time. Practice tests also provide useful and timely feedback to both students and teachers so that they may adjust their revision or teaching strategies.

From a learning and retention point of view, short-answer tests are more effective than multiple-choice tests [13]. This argument is supported by a recent research study [14]. The study shows that short-answer questions (that require recall) are more beneficial than multiple-choice questions (that require recognition) for subsequent memory performance. The study also reveals that taking a practice test is a more potent learning device than additional study of the target material and it leads to better performance on final test. In addition, experimental results also demonstrate that short-answer tests produce greater gains in student performance on final test than multiple-choice tests. These findings provide a good rationale for researching in the area of automated short-answer marking so that students' learning and retention of course content can be improved.

The short-answer marking system presented in this paper is called *IndusMarker*. IndusMarker exploits *structure matching*, i.e., matching a prespecified structure, developed via a purpose-built *structure editor* [15], with the content of the student's answer text. The examiner specifies the required structure of an answer in a simple purpose-designed language. The language was initially called QAL but later on the authors redefined it as a sublanguage of XML and named it Question Answer Markup Language (QAML). This section presents QAL and the next section is about QAML.

The syntax and semantics of QAL is intended to be suitable for educators with widely differing computing skills, i.e., QAL is intentionally simple enough to be readily understandable and hence easy to learn. The language also embodies the necessary constructs to express a structure for a natural language text. To get an overview of how the required structures are written in QAL, consider the following example:

Test question: Describe the idea of "function overloading" in a single sentence. *(1 mark)*

Model answer: Functions having the same name but different signatures.

The examiner must specify the required structure so that all the expected *paraphrases* are elaborated. There are two equally important required components or *regions* in the model answer for the above question. The first required region is "same name" and the second is "different signatures." Since both are of equal importance, they are allocated equal marks. The following is the *regions specification* in QAL for the above question:

```
Begin_regions;
      Begin_region(marks = 0.5);
            "same name"
      End_region;
      Begin_region(marks = 0.5);
            "different signature"
      End_region;
End_regions;
```

Each region has its own required structure which consists of multiple possibilities (each possibility representing the structure of a possible paraphrase for the region) as shown below:

Region: "same name"

Possibility #1:
```
<main> = ("same";"name")*2:0.5:;
```

Possibility #2:
```
<main> = ("same";"identity")*2:0.5:;
```

Region: "different signature"

Possibility #1:
```
<main> = ("different";"data type")*2:0.5:;
```

Possibility #2:
```
<main> = ("different";"parameter")*2:0.5:;
```

Possibility #3:
```
<main> = ("different";"signature")*2:0.5:;
```

Possibility #4:
```
<main> = ("different";"argument")*2:0.5:;
```

The examiner uses a set of previous students' answers together with the model answer to predict all acceptable paraphrases for a region. The following is a brief explanation of the constructs used in the possibilities specification of a region:

1. **Notation:** +
   Meaning: Sequence
   Example: `"more"+"one"`
   Explanation of the example: In the student answer text, the word "more" should appear before "one."

2. **Notation:** `NP_containing(.....)`
   Meaning: Noun phrase containing any of the strings specified in the enclosed brackets.
   Example: `NP_containing("same type" | "same data type")`
   Explanation of the example: Noun phrase containing either `"same type"` or `"same data type."`

3. **Notation:** `VG_containing(.....)`
   Meaning: Verb group containing any of the strings

specified in the enclosed brackets.
Example: `VG_containing("facilitate" | "assist" | "ease" | "help")`
Explanation of the example: Verb group containing either `"facilitate"` or `"assist"` or `"ease"` or `"help."`

4. Notation: [.....]
Meaning: Condition. The three allowed conditions are `NO_WORD`, `NO_NP`, and `NO_VG`. These conditions, respectively, mean "no word," "no noun phrase," and "no verb group" allowed at a particular location in the student answer text.
Example: `<DBMS>+[NO_VG]+<organize>+[NO_NP & NO_VG]+"data"`
Explanation of the example: There should be no verb group between the subpatterns `<DBMS>` and `<organize>` and there should also be no noun phrase as well as no verb group between the subpattern `<organize>` and the word `"data."`

5. **Notation:** {...}
Meaning: Alternative options. Any one of the alternatives.
Example #1: {`"count", "counting"`}
Explanation of the example: Any one of the two alternatives.
Example #2: {`"type int","type"&"int"`}
Explanation of the example: Either the string `"type int"` or the string `"type"` followed by the string `"int"` (there may be some other string between `"type"` and `"int"`).

6. **Notation:** (...) `*MinNum`
Meaning: A specified minimum number of options should appear in the student answer text (in any order).
Example: `(<organization>;"storage;" "access";"security";<integrity>)*3`
Explanation of the example: At least three of the five options should appear in the student answer text.

7. **Notation:** `NOT(...)`
Meaning: The word(s) contained in the NOT word list is/are not allowed at a particular location in the student answer text.
Example:
`NP_containing("array")+NOT("structure")+ NP_containing("same type" | "same data type")`
Explanation of the example: The word `"structure"` should not appear between the noun phrase containing `"array"` and the noun phrase containing `"same type"` or `"same data type."`

Reconsider the following possibility of the "same name" region from the example presented earlier:

$$\overbrace{\texttt{<main>}}^{\text{A}} = \overbrace{(\text{“}\texttt{same}\text{”};\text{“}\texttt{identity}\text{”}) * 2}^{\text{B}} : \overbrace{0.5}^{\text{C}} ;;$$

An explanation of the marked parts of the possibility is given below:

"A" indicates that this is the main pattern of the possibility. A possibility can also have subpatterns called from the `"main"` pattern. This is analogous to the concept of `"main"` method in Java where program execution starts from the main method but other methods can be called (invoked) from the main method.

"B" indicates that at least the two options should appear in at least one of the student answer sentences (in any order). Since there are only two options (i.e., `"same"` and `"identity"`), both should appear in at least one of the student answer sentences.

"C" indicates the marks that will be added to the student's total if this structure is found in any sentence of the student answer text.

4 The Question Answer Markup Language (QAML)

XML is currently the de facto standard format for data handling and exchange [16]. Another advantage of XML is that the data in XML documents are self-describing. Customized markup languages can also be created using XML and this represents its direct utility. In other words, XML is a metalanguage and has the ability to define new languages built around a standard format [17], [18]. People, who create these new languages, can also tailor them to their own specific needs. The authors, therefore, decided to redefine QAL as a sublanguage of XML so that QAL is standardized and all the benefits of XML can be exploited. The new language is called QAML.

The authors had to first build rules that specify the structure of a QAML document so that the document can be checked to make sure it is set up correctly. There are two types of QAML documents: one type contains "regions specification" for expected answers and the other type contains "possibility specification" for a region. Since each region may have multiple possibilities, there may be more than one "possibility specification" document for a region.

To have a better understanding of how QAML represents the required answer structure for a question, consider the following example:

Test Question: Explain the meaning of the following C++ statement: `int *m[10];` *(1 mark)*

Model Answer: The C++ statement represents declaration of an array of pointers to integer variables. The size of the array is 10.

The following is a possible "regions specification" of the required answer structure for the question:

```
<QAML_REGIONS_SPECIFICATION>
  <BODY>
    <REGION>
      <TEXT>Declaration of an array of
      pointers to integer variables, size
      10</TEXT>
     <MARKS>1</MARKS>
    </REGION>
  </BODY>
</QAML_REGIONS_SPECIFICATION>
```

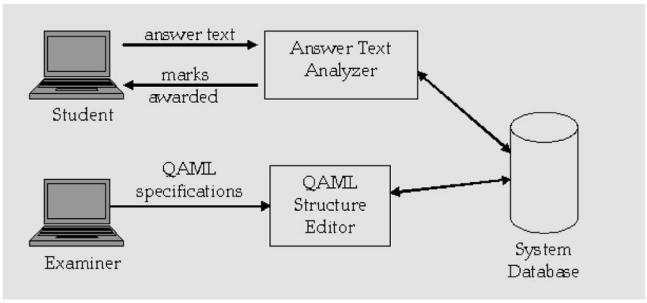


Fig. 1. An overview of the system architecture.

There is only one region in the above QAML "regions specification." Some other person designing the required structure may decide to have a different "regions specification," e.g., s/he may decide to have more than one region in the "regions specification." The decision about how many regions to create in a "regions specification" depends mainly on the total number of concepts expected to appear in the student answer text and the maximum marks for the question. A designer can choose to have more than one concept in a region. S/he also has to make a decision about the region marks and this depends upon the relative importance of the particular region. The following are the QAML "possibility specifications" for the first two possibilities of the region in the "regions specification:"

Possibility #1:

```
<QAML_POSSIBILITY>
    <MAIN_PATTERN>
        <PATTERN_BODY_PART>
            <SEQUENCE>
                <TEXT>array</TEXT>
                <TEXT>of</TEXT>
                <TEXT>pointer</TEXT>
            </SEQUENCE>
            <MARKS>1</MARKS>
        </PATTERN_BODY_PART>
    </MAIN_PATTERN>
</QAML_POSSIBILITY>
```

Possibility #2:

```
<QAML_POSSIBILITY>
    <MAIN_PATTERN>
        <PATTERN_BODY_PART>
            <SEQUENCE>
                <TEXT>pointer</TEXT>
                <CONDITION>
                    <NO_WORD/>
                </CONDITION>
                <TEXT>array</TEXT>
            </SEQUENCE>
            <MARKS>1</MARKS>
        </PATTERN_BODY_PART>
        <PATTERN_BODY_PART>
            <SEQUENCE>
                <TEXT>pointer array</TEXT>
            </SEQUENCE>
            <MARKS>1</MARKS>
        </PATTERN_BODY_PART>
    </MAIN_PATTERN>
</QAML_POSSIBILITY>
```

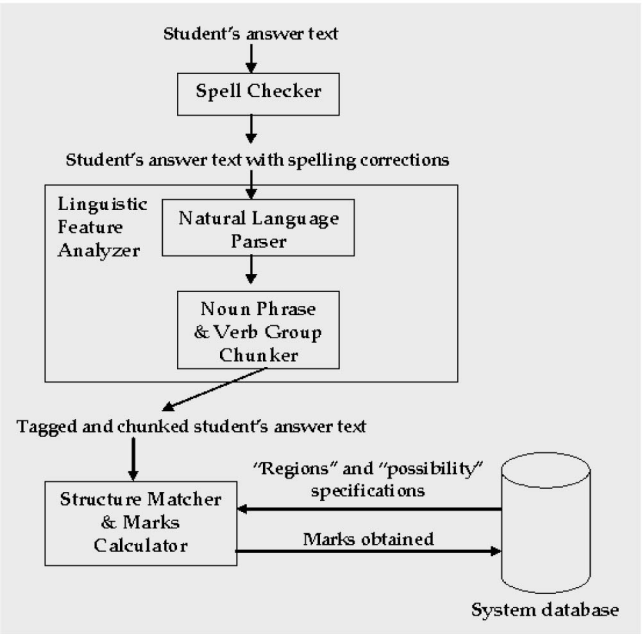


Fig. 2. Architectural design of the "Answer Text Analyzer."

All the above QAML specifications have to be well formed and valid. In order for these specifications to be valid, they have to comply with the DTD associated with them. IndusMarker has a QAML structure editor that helps the user build well-formed and valid QAML specifications.

5 Architectural Design of the System

IndusMarker can be roughly divided into two main components: an "answer text analyzer" and a "QAML structure editor." The system is designed for two types of users: examiners and students. Interactions between the main components and users of the system are depicted in Fig. 1.

5.1 Answer Text Analyzer

This component of IndusMarker has three subcomponents: a spell checker, a "linguistic feature analyzer," and a "structure matcher and marks calculator." The "linguistic feature analyzer" itself has two further subcomponents: a "natural language parser" [19] and a "noun phrase and verb group chunker." Architectural design of the "answer text analyzer" is depicted in Fig. 2. An "answer text analyzer" performs four functions:

1. spell checking,
2. some basic linguistic analysis—Part-Of-Speech (POS) tagging and Noun Phrase and Verb Group (NP and VG) chunking,
3. matching student's answer text structure with the required structure (as specified in the "regions" and "possibility" specifications), and
4. computing the total marks of the student for their answer based on the result of the matching process.

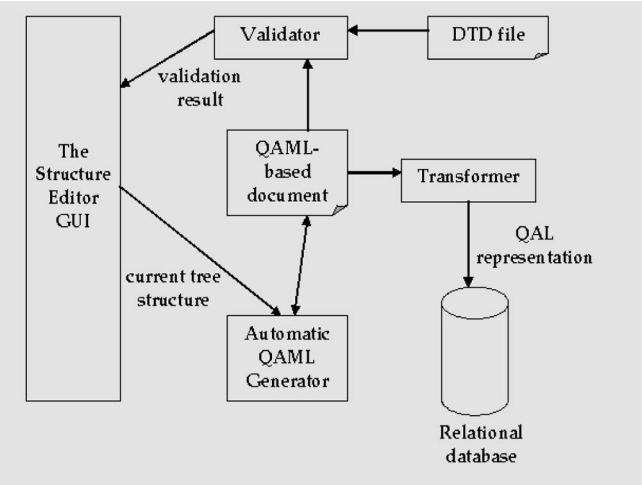


Fig. 3. Architectural design of the "QAML Structure Editor."

The spelling mistakes in the student's answer text are highlighted by the spell checker and correct spelling options are presented to the student for each spelling mistake. It is the responsibility of the student to make the final decision about the correct spelling. Once the student has submitted an answer text, some basic linguistic analysis is performed. A natural language parser and NP and VG chunker are, respectively, used to perform POS tagging and NP and VG chunking of the student's answer text. After linguistic analysis, the tagged and chunked student's answer text is processed by the "structure matcher and marks calculator." The result of structure matching is used to compute the total marks obtained by the student for their answer. The technology used in the "Answer Text Analyzer" is described more thoroughly in Section 6.1.

5.2 QAML Structure Editor

The "QAML structure editor" performs four important functions:

1. provides a suitable Graphical User Interface (GUI) that enables development of structured QAML specifications with relative ease,
2. automatic QAML document generation,
3. validation of QAML documents by ensuring that all rules in the associated DTD are followed, and
4. transformation of QAML specifications to their respective QAL representations. (The reason for this transformation function is that the "answer text analyzer" was designed and implemented for QAL and the idea of QAML came later. Since the "answer text analyzer" had already been developed and tested, there was no need to make any change in this component if the QAML structures can somehow be transformed to QAL equivalents.)

The architectural design of the "QAML structure editor" is depicted in Fig. 3.

The structure of QAML specification is represented at the user interface as a tree. Nodes may be added or deleted from this tree. Each node represents a QAML element. A leaf node represents either an empty QAML element or parsed character data of a QAML element. Changes in the tree structure result in corresponding changes in the QAML document. The contents of the QAML document are serialized by an "automatic QAML generator." A validator is used to verify whether the QAML document conforms to all the constraints specified in DTD. Once the QAML document has been validated, the QAML specifications are transformed to their QAL representation for storage in a relational database (see Fig. 5 for the internal structure of the transformer). The technology used in the "QAML structure editor" is described more thoroughly in Section 6.2.

6 The System Implementation

The system implementation can be subdivided into implementation of the two major system components (i.e., the "answer text analyzer" implementation and the "QAML structure editor" implementation). A number of technologies, including Java, XML, DTD, XSL/XSLT, Simple API for XML (SAX), XPath, the Stanford Parser, JOrtho, etc., are used in the system implementation. The following sections describe how these technologies are utilized in the system's development.

6.1 The "Answer Text Analyzer" Implementation

The "answer text analyzer" component consists of three subcomponents: the "spell checker," the "linguistic feature analyzer," and the "structure matcher and marks calculator." The spell checker used is called JOrtho (Java Orthography) [20]. It is an Open Source spell checker and is entirely written in Java. Its dictionaries are based on the free Wiktionary project [21]. The JOrtho library works with any JTextComponent from the Swing framework. In the case of IndusMarker, the spell checker is registered with JTextPane component in which the student's answer is supposed to be entered. The JOrtho library, when bound to a JTextComponent (such as JTextPane), highlights the potentially incorrectly spelt word and offers a context menu with suggestions for a correct form of the word. The students, entering their answer, must select a correct spelling before submitting their answer because incorrect spelling will not be automatically corrected once the answer has been submitted.

To develop the required linguistic feature analysis capability, the authors have used the Stanford Parser [19] (as a natural language parser) and a self-developed Noun Phrase and Verb Group (NP and VG) chunker. The Stanford Parser is a program that determines the grammatical structure of sentences. It is a Treebank-trained statistical parser developed by Klien and Manning at Stanford University and is capable of generating parses with high accuracy [22]. Key reasons for choosing the Stanford Parser were: 1) it is written in Java and since the rest of the system software is also written in Java, the parser is easy to integrate with the system, and 2) the parser is highly accurate. The parser can read plain text input and can output various analysis formats, including part-of-speech tagged text, phrase structure trees, and a grammatical relations (typed dependency) format. In the case of the author's system, the Stanford Parser is used only to get the part-of-speech tagged text output. The tagged text output is

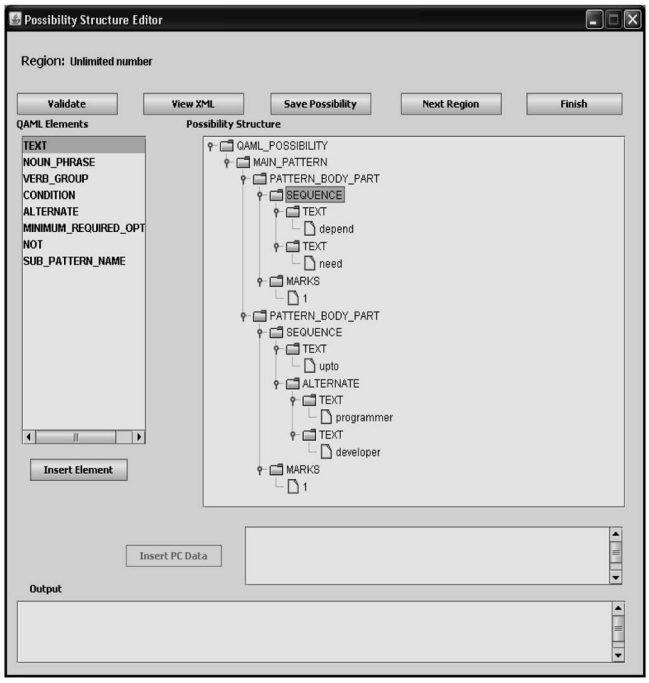


Fig. 4. Screen shot of the "possibility structure editor" GUI.

then chunked into noun phrases and verb groups by a self-developed NP and VG chunker.

The "structure matcher and marks calculator" subcomponent is designed by the authors themselves. Its main function is to compare the structure of the student's answer text with the structure specified in the related QAL specification. Based on the result of such a comparison, the student's marks for the answer are computed.

6.2 The "QAML Structure Editor" Implementation

The GUI of the "QAML structure editor" enables users of the system to exploit the necessary functionalities more easily. Fig. 4 is a screen shot of the "QAML structure editor." Since QAML specifications are divided into two parts (i.e., "regions" and "possibility structure" specifications), the GUI of the "QAML structure editor" consists of two screens: one for specifying the "regions" structure and the other for specifying the "possibility structure." The screen in Fig. 4 is the GUI for the "possibility structure" specification. The GUI for the "regions" specification is similar to the GUI for "possibility structure" specification. The QAML specification is represented as a tree structure in the GUI. The functionalities of the structure editor are implemented using various XML-related technologies.

6.2.1 The "Validation" Functionality Implementation

The QAML document validation functionality is implemented using the SAX [23]. SAX enables Java programs to parse and validate XML documents.

6.2.2 The "View XML" Functionality Implementation

The "View XML" functionality enables users of the structure editor to dynamically view the current status of their QAML documents. The XML Document Object Model (DOM) is used to implement this functionality. The principle behind programming with DOM is simple. The first stage involves using a parser to translate the XML document into an in-memory tree of objects representing every element and attribute. The methods in these objects' interfaces can be used to navigate around the document, extract information, and modify these objects' content. The in-memory object hierarchy can also be converted back to XML [24].

In order to use DOM, the DOM parser needs to be first accessed, and this is done by code that is entirely proprietary to Xerces. The parser is implemented as a `DocumentBuilder object.` This is because what it actually does is build a document object model from the incoming data source, via the method `parse()`, which returns a `Document` object [24].

The `Document` object contains the in-memory object hierarchy. The "View XML" functionality requires that the structure and content of the GUI tree in the structure editor should be the same as the in-memory DOM-based tree. The in-memory tree is likely to be different from the GUI tree. In order to make sure that the two trees are synchronized and the updated QAML document is displayed, the following steps are taken: 1) the child nodes of the in-memory tree root node are deleted, 2) the in-memory tree is repopulated based on the current structure and content of the GUI tree, and 3) the in-memory tree is then written back to the QAML document.

After the updated in-memory tree has been written back to the QAML document file, the contents of the QAML document file are then displayed in a separate GUI screen.

6.2.3 The "Save QAML Specification" Functionality Implementation

The QAML-based specifications (both "regions" and "possibility") need to be stored in a database. An important point to remember here is that IndusMarker was initially built for QAL and not for QAML. The system had to be adapted for this important change. A design decision was made that there was no need to change the "answer text analyzer" component. The reason was that the logic and working of the "answer text analyzer" had already been tested with QAL and if somehow the QAML-based specification was transformed to the QAL-based equivalent and this QAL-based equivalent was stored in the system database (instead of the QAML-based specification), then no modification to the "answer text analyzer" component was required. The transformation of the QAML-based specification to an equivalent QAL-based specification is achieved using the XSL/XSLT and the XPath technologies. The QAL-based specification is then stored in a relational database. The subcomponents involved in this transformation and storage process are depicted in Fig. 5.

XSL is a language that transforms a document from one format to another [23]. In the case of IndusMarker, the XSL style sheet is used to transform a QAML-based specification to an equivalent QAL-based specification. The Apache Xalan XSLT processor takes in the XSL style sheet and the QAML-based document as input and produces an HTML document containing a QAL-based specification. Since the QAL-based specification is in the HTML document, it needs to be extracted and stored in the relational database of the

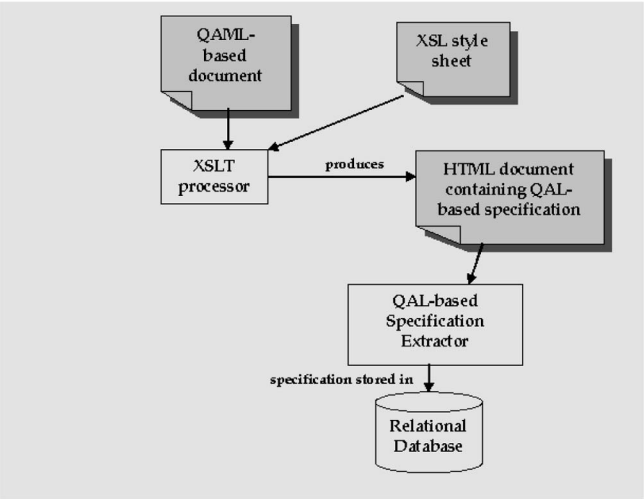


Fig. 5. Subcomponents involved in the transformation and storage process.

system. This task is carried out by a "QAL-based Specification Extractor." The coding details of the style sheets and the specification extractor are beyond the scope of this paper. The next section presents the types of short-answer questions that are suitable for marking by the system.

7 Allowed Short-Answer Question Types

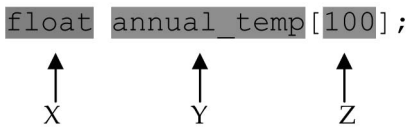
IndusMarker cannot currently mark all types of short-answer questions with a high degree of accuracy. The limitations of the system must be defined and one means of doing so is to clearly define the types of short-answer questions that the system is designed to process. The following is the list of OOP short-answer question types that are expected to be marked by the system with a satisfactory degree of accuracy (the question types are listed in increasing order of complexity):

1. "True" or "false" question: This type of question requires student to state whether a particular statement is "true" or "false." An example of such a question is given below:
   State whether the following statements are true or false:
   - *A class is an instance of an object.*
   - *The destructor of a class never has any arguments.*
2. Sentence completion: This type of question requires student to supply the missing words in an incomplete sentence. An example of such a question is given below:
   - *The wrapping up of data and member function into a single unit is called _____.*
   - *If there is a pointer* `p` *to objects of a base class, and it contains the address of an object of a derived class, and both classes contain a nonvirtual member function, ding(), then the statement* `p->ding();` *will cause the version of* `ding()` *in the _____ class to be executed.*
3. Single term generation: This type of question requires students to generate a single term. The student's answer may be longer than the required term, but it is the required term that the assessor is looking for in the answer. Two examples of such questions are:
   - *What is the name of a special member function that has no return type and has the same name as the name of the class?*
   - *Data abstraction and inheritance are key features of Object-Oriented Programming (OOP). Name one more key feature of OOP.*
4. "Quantity" required: This type of question normally starts with the words "how many" and requires students to specify some quantity. An example of a "quantity required" question is given below:
   Consider the following piece of code (of Java programming language):
   ```
   int x = 3;
   for(int i = 0;i<x;i++)
   System.out.println("AAA");
   ```
   How many times the above for loop will iterate?
5. "Numerical value" generation: This type of question requires the generation of a numerical value. An example of such a question is given below:
   Consider the following C++ statement:
   ```
   int primes[] = {1,2,3,5,7,11,13};
   ```
   What is the size of the array primes declared in the above C++ statement?
6. "Location" required: This type of question requires students to identify a particular location, e.g., a part of a text. An example of such a question is given below:
   - *Where do C++ programs begin to execute?*
   - *In C++, where do we place the* `virtual` *keyword to indicate that a function is virtual?*
7. "Program statement output" required: This type of question requires students to generate the output of a print statement contained in a program. Through this the examiner can test student's understanding of the program. An example of such a question is given below:
   Consider the following C++ piece of code:
   ```
   #include <iostream>
   using namespace std;
   int main(){
          int i = 10;
          int *m = &i;
          int n = *m+*m;
          cout << m << "\n";
          cout << n << "\n";
   }
   ```
   What will get printed on the console due to the second cout statement?

The student taking the test can only answer the question if s/he has good understanding of the "address of" (&) and "value of" (*) operators.

8. Single phrase generation: This type of question should be answerable through a single phrase although the answer supplied by a student may be longer. Examples of such questions are:

   Consider the following C++ statement:

   The above C++ statement is a declaration of an array and three of its syntactical parts are highlighted and named as X, Y, and Z.

   - *What does part X represent?*
   - *What does part Y represent?*
   - *What does part Z represent?*

9. "Example" required: This type of question requires students to provide examples of a given term, situation, etc. The number of possible examples must be finite and easily predictable in advance because it is impossible to correctly mark valid, unpredictable examples. Examples of questions of this type are:

   - *Give an example of a relational operator?*
   - *Give an example of an arithmetic unary operator?*

10. List: This type of question requires student to specify reasons, constructs, items, entities, etc., that fall under a particular category or satisfy specific conditions. Examples of questions of this type are:

    - *List three types of loops available to a C++ programmer.*
    - *Every software object has two characteristics. Name these two characteristics.*

11. Short explanation/description: This type of question requires students to provide a short explanation or description of some statement(s), process, or logic behind a piece of code, etc. The explanation or description should ideally be no more than two sentences long. Examples of questions of this type are given below:

    - *Describe the idea of "function overloading" in a single sentence.*
    - *Explain the meaning of the following C++ statement:* `int *m[10];`

12. "Situation" or "context" required: This type of question normally starts with "when" and requires students to specify the situation or context in which a particular condition is valid or an event occurs. Two examples of such questions are:

    - *When does C++ create a default constructor?*
    - *When do we use the keyword* `virtual` *in C++?*

13. Definition: This type of question requires students to provide a definition of a specified term. The term must be definable in one or two sentences, however, a student can provide a longer definition. Examples of such questions are:

    - *What is a ternary operator?*
    - *What is a variable?*

14. Contrast: This type of question requires students to identify differences between two things. Two examples of such questions are:

    - *What is the main difference between a while and a do while statement?*
    - *What is the difference between private and public members of a class?*

    When designing this type of question, it must be remembered that the number of possible differences is small (ideally not more than two or three in number). This restriction on the number of differences is necessary in order that the possible difference(s) can be easily predicted by the examiner. The examiner can then specify the required structure based on his/her knowledge of possible difference(s).

15. Compare: This type of question requires students to state a similarity between two things. Two examples of such questions are:

    - *What is the most important similarity between C++ and Java programming languages?*
    - *What is the similarity between a base class object and a derived class object?*

    The number of possible similarities must again be small and easily predictable by the examiner in advance.

16. Composite questions: A question that consists of more than one part with each part itself a question of one of the question types 1-15 above. Two examples of such questions are:

    - *What is a ternary operator? Give an example of a ternary operator.*
    - *What is the purpose of* `delete` *operator? What is called when the* `delete` *operator is used for a C++ class object?*

8 Evaluation and Analysis

IndusMarker has been evaluated by assessing students' answers to purpose-designed tests. The first task was to design tests suitable for IndusMarker to mark. The lecturers involved were informed about the types of short-answer questions that IndusMarker can mark. Six OOP tests were designed so that the performance of IndusMarker can be satisfactorily evaluated on all types of short-answer questions. Two hundred and twenty five students of BU Karachi campus undertook each of the six OOP tests. Two lecturers and six teaching assistants of BU Karachi campus took part in the evaluation process. Teaching assistants carried out manual marking of students' answers while lecturers performed the required structure formulation and validation. Both lecturers and teaching assistants had reasonably good knowledge of OOP concepts. The lecturers were

TABLE 1
Summary of the System's Performance on All the Allowed Short-Answer Question Types

Question type	Number of questions used	Average answer length (words)	Average human-system agreement rate
"True" / "False" question	8	1.2	100%
Sentence completion	6	1.5	99.67%
Single term generation	5	1.9	99%
"Quantity" required	7	3.1	99.5%
"Numerical value" generation	3	2.3	98.83%
"Location" required	3	3.6	99.16%
"Program statement output" required	12	3.9	98.79%
Single phrase generation	5	7.3	93.7%
"Example" required	3	5.6	99.33%
List	4	7.1	92%
Short explanation / description	6	11.6	92.33%
"Situation" or "context" required	5	11.2	93.5%
Definition	4	10.8	93.67%
Contrast	5	20.4	84.1%
Compare	6	15.6	94.12%
Composite questions	5	14.9	92%

provided with guidelines on how to write required structures in QAML using the systems' structure editor.

The system was made available online by a software developer at BU. The students' answers were collected online and first marked manually by teaching assistants. The students' answers for each question were divided into two parts: 25 students' answers were kept for the required structure formulation and the remaining 200 students' answers were kept for the required structure validation. To define the required structure for a question, a lecturer analyzed the structure and content of the model answer and 25 students' answers. Once the required structure for a question had been developed using the system's structure editor, the required structure was tested using the remaining 200 students' answers. Table 1 summarizes the structure testing results for all the allowed short-answer question types presented in Section 7. The table shows the number of questions used, the average answer length and the average human-system agreement rate for each question type.

In order to interpret data in Table 1, it is important to understand how the values presented have been calculated. The distinction between a question and a question type is important to remember. Multiple questions were used for a particular question type. For example, five questions were used for the "single term generation" question type. To calculate the average answer length (in words) for a question type, the authors used the following formula:

$$\mathrm{y} = \frac{\sum_{i=1}^{q} w_{\mathrm{i}}}{q \times a},$$

where y is the average answer length (in words), w_{i} is the total number of words in all the answers for ith question, q is the total number of questions used for the question type, and a is the total number of answers per question.

To calculate the average human-system agreement rate for a question type, the human-system agreement rate for each question belonging to the question type is first calculated:

$$\mathrm{r} = (\mathrm{a}/\mathrm{t}) \times 100,$$

where r is the human-system agreement rate for the question, a is the number of judgments where the human and system agree, and t is the total number of judgments.

The average human-system agreement rate for the question type is then calculated using the following formula:

$$z = \frac{\sum_{i=1}^{q} r_i}{q},$$

where z is the average human-system agreement rate, $\mathrm{r_i}$ is the human-system agreement rate for the ith question, and q is the total number of questions used for the question type.

Table 1 demonstrates that if an OOP short-answer question test is carefully designed, high human-system agreement rates can be achieved. The average human-system agreement rate tends to decrease as the complexity of the short-answer question type increases. This trend is not valid across all question types. For example, "compare" questions are deemed to be more complex than the "single phrase generation" questions by the authors but the average human-system agreement rate is higher for the "compare" questions than that for the "single phrase generation" questions. "Contrast" questions have the lowest average human-system agreement rate while "true"/"false" questions have the highest average human-system agreement rate. Another important pattern that may be deduced from the data in the table is that (in general) as the average answer length increases the average human-system agreement rate decreases but again this is not true in every case. The "contrast" question type has the highest average answer length and the lowest average human-system agreement rate. But the "example required" question type has both higher average answer length and higher average human-system agreement rate than the "single term generation" question type.

The errors of IndusMarker were analyzed and they fall into two categories: *misses* and *false positives*. A miss occurs when a response gets lower marks than it deserves. A false positive occurs when the system assigns more marks to a response than it deserves. In case of the system's evaluation, the number of misses was much higher than the number of false positives. Around 69 percent of all the errors were misses while only 31 percent of the errors were false

positives. The relatively higher ratio of misses is due to the fact that it is very difficult to anticipate all the possible paraphrases for an answer. If some correct possibility is missed out by the person specifying the required structure, then the occurrence of that possibility in students' answers will lead to misses.

There are two reasons for the system's false positives. The first is when a student does not know when to stop typing—beginning with a correct answer but going on to say something that is clearly wrong. It is impossible to predict all the wrong possibilities in advance. So, the required structure specification normally contains correct possibilities only. The system assigns marks when it finds the correct possibility that it is looking for. It does not normally search for wrong parts of the students' answers. The second reason for false positives is that the student happens to use the correct language—but that the language is used in such a manner that it does not, in fact, convey the correct concept.

If the human-system agreement rate for a question is not 100 percent, the required structure may be modified by analyzing the structure and content of students' answers where there is discrepancy between human and system markings. Once the required structure for a question has been finalized, it is stored for future use.

These tests were designed to be used as low-stake, practice tests. Since IndusMarker cannot guarantee a 100 percent human-system agreement rate, it cannot be used for high-stake tests. The approach to using the system for practice tests is only effective if the same practice test is repeated many times. The lecturers at BU indicated that in most cases the curriculum of a course does not change for several years, and therefore, the same practice test may be used for many terms or semesters. When a practice test is taken for the first time, the students' answers are manually marked and these manually marked students' answers are used to develop and validate the required structures. The required structures are stored in the system's database and can be used to mark future practice tests. This is similar to the situation where a lecturer spends a considerable amount of time preparing a lecture presentation in the form of PowerPoint slides and then reuses the same lecture slides for several years. In this way, time spent preparing the slides when a course is taught for the first time is compensated for if the same course material is taught for several years and/or by many different lecturers. In fact, effort expended and time spent initially results in much greater time and effort being saved later. Similarly, if some time and effort is consumed making and validating the required structures for a practice test, then this results in much greater benefit later on if the same test is repeated. Since the marks obtained in these practice tests do not contribute to the final grade of students, and the main objective is to promote learning and provision of immediate (and accurate) feedback to both students and teachers, it is expected that students will not raise serious objections even if there is some lack of trust in the system's marking accuracy.

In order to be confident about the feasibility of the system, it is also important to consider whether the lecturers find it easy to learn QAML and to use the system. The lecturers were given a 15-minute presentation about the QAML and how to use the system. The lecturers found QAML a simple and a sufficiently expressive language through which the required structures can be expressed conveniently. According to the lecturers' comments, the system is easy to use and the QAML structure editor is quite helpful in the task of required structure specification. Table 2 shows the time taken by the lecturers to formulate and validate the required structures for the six OOP tests used in the evaluation. The lecturers found the time consumed quite reasonable and manageable given that these tests will be repeated many times and their automated marking will provide useful benefits to both students and teachers. Time taken to formulate and validate the required structures for a particular OOP test depends upon the number and complexity of the questions appearing in that test.

TABLE 2
Time Taken to Formulate and Validate the Required Structures for Each of the Six OOP Tests

Test	Time taken
First OOP test	1 hour 50 minutes
Second OOP test	2 hours 10 minutes
Third OOP test	2 hours 45 minutes
Fourth OOP test	1 hour 35 minutes
Fifth OOP test	1 hour 40 minutes
Sixth OOP test	2 hours 35 minutes

Once the required structures for questions appearing in the six OOP tests were finalized (i.e., when the process of required structure formulation, validation, and correction had ended), the required structures were stored in the system database. The required structures were then used when these tests were later given to students studying the same course at BU Islamabad campus. The tests were conducted online in computer-based classrooms, and students who took these tests obtained summative feedback on their performance in the form of marks immediately after the test. Marks obtained in each question and total marks obtained in the test were provided. If a student thinks that the marks given by the system are not accurate then the student has the option to view the correct model answer so that he or she can compare his or her own answer with the correct model answer. Moreover, the system can highlight the parts of the student's answer text that have been matched with the required structure. It can also indicate to the student the regions (of the QAML regions specification) that have been found and those that have not been found in the student's answer text. In this way, the student can have a better idea of how his or her marks for a particular answer have been calculated. He or she can also better understand his or her mistakes.

Lecturers can also view the results of the test if they log in to the system. Lecturers can then revise those topics where the overall students' performance is poor, or give additional tutorials to those students who are performing poorly. Students can also get an idea of their overall understanding of the course content and can increase their learning effort if their performance is poor. Both students and lecturers at the BU Islamabad campus found the system quite useful and its performance satisfactory. The system can also be viewed as a tool that promotes "deep learning"

[25], [26]. In such learning, both lecturer and students actively participate in students' learning. The lecturer needs to obtain feedback on their teaching performance from students, and students need feedback from the lecturer on their learning performance. The feedback must reach the students as quickly as possible in order to affect their learning, that is, to promote deep learning. This objective can be achieved through proper use of the system.

9 Similar Automated Short-Answer Marking Systems

Before concluding the paper, a brief overview of similar automated short-answer marking systems is presented in this section. Later on in this section the authors state the advantages of IndusMarker over other similar systems.

9.1 C-Rater

C-rater is an automated short-answer marking engine developed by Education Testing Service (ETS) [2], [27]. It is designed to score factual answers, and therefore, the number of possible correct answers expected from students is finite. If there is a set consisting of all the possible correct student responses, then the C-rater scoring engine operates as a *paraphrase recognizer* that identifies members of this set.

A model of the correct answer has to be created by a "content expert." C-rater's task is to map the student's response onto this model and, in so doing, check the correctness of the student's response. Before this mapping can take place, the student's response is first converted to a *canonical representation* (i.e., a nonambiguous, mutually exclusive representation of "knowledge") by C-rater.

C-rater has been evaluated in two large-scale assessment programs [2], [27]. The first was the National Assessment of Educational Progress (NAEP) Math Online Project. C-rater was used to evaluate written explanations of the reasoning behind particular solutions to some Maths problems. Five such questions were used in the evaluation process. The second program was the online scoring and administration of Indiana's English 11 End of Course Assessment pilot study. In this case, C-rater was required to assess seven reading comprehension questions. The answers to these questions were more open-ended than those to the questions in NAEP Math Online Project. In NAEP assessments, the average student response was around 15 words or 1.2 sentences long. Each student response was scored by C-rater and scored separately by two human judges. Here, 250 to 300 student responses were used for each question. The agreement percentage between C-rater and the first human judge was 84.4 percent while between C-rater and the second human judge was 83.6 percent.

In the Indiana pilot study, student responses were longer and the average length was around 2.8 sentences or 43 words. One hundred student responses were used for each question and were scored separately by C-rater and a human judge. Leacock and Chodorow [2] summarized the evaluation results: "On average, C-rater and the human readers were in agreement 84 percent of the time."

9.2 The Information Extraction (IE) Based System Developed by Sukkarieh and Pulman [3]

The IE-based short-answer marking system was developed at Oxford University to fulfill the needs of the University of Cambridge Local Examination Syndicate (UCLES). The system depends on pattern matching for the calculation of marks. A human expert discovers information extraction patterns. A set of patterns is associated with each question. This set is further divided into bags or equivalence classes. The members of an equivalence class are related by an equivalence relation, i.e., a member of an equivalence class conveys the same message and/or information as other members of the same equivalence class. The marking algorithm compares student answers with equivalence classes and awards marks according to the number of matches.

The evaluation of the latest version of the system was carried out using approximately 260 answers for each of the nine questions taken from a UCLES GCSE biology exam. The full mark for these questions ranged from 1 to 4. Two hundred marked answers were used as the training set (i.e., the patterns were abstracted over these answers) and 60 unmarked answers were kept for the testing phase. The average percentage agreement between the system and the marks assigned by human examiner was 84 percent [3].

9.3 Automark

Automark has been developed for robust automated marking of short free-text responses [4]. IE techniques have been used to extract the concept or meaning behind free text and full effort has been made to make the software system tolerant of errors in typing, spelling, syntax, etc. Automark uses mark scheme templates to search for specific content in the student answer text. These templates are representatives of valid (or specifically invalid) answers. The templates are developed using an offline custom written configuration interface. The software system first parses the student answer text and then "intelligently" matches it with each mark scheme template so that marks for the student answer may be calculated. The answer representation of a mark scheme template may be mapped to a number of input text variations.

Automark was tested in a real world scenario of "high importance" tests that were part of UK national curriculum assessment of science for pupils at age 11 [4]. Four items (of varying degrees of linguistic complexity) were taken from 1999 papers. For each item, 120 student answers were taken. The Automark system was able to correctly mark most of the answers containing spelling, syntax, and semantic errors. The system was able to show its "understanding" of badly expressed ideas and its marking co-relation with human marking ranged between 93.3 and 96.5 percent.

9.4 Advantages of IndusMarker over Other Similar Systems

It is important that IndusMarker is compared with other similar systems so that the contribution made may be highlighted. Advantages of IndusMarker over other similar systems are:

1. IndusMarker is based on a new, extensive, and formal language called QAML to express required answer structures. The other systems lack such an extensive and well-designed language.
2. QAML is defined as a sublanguage of XML. XML is currently the standard format for data handling and exchange [16]. The other systems did not exploit the utility of XML.

3. A well-designed structure editor based on state of the art XML-related technologies such as SAX, DOM, XSL, etc., has been created. The structure editor helps users specify QAML-based specifications. No such serious effort was made in the case of the other three systems.
4. An important QAML concept is the use of a "regions" specification. The required answer structure for longer, multipart answers can be easily represented if the expected answer text is considered to consist of various regions. So, an important aspect of the system's capability is the ability to process "composite questions" structured from other "composite" or "simpler" questions. In this way, IndusMarker can mark longer, factual answers. This feature is lacking in the other three systems.
5. The allowed short-answer question types are clearly stated in the case of the authors' system. This means the domain and scope of the authors' system is more clearly defined than the other three systems.
6. A reasonably good human-system agreement rate is achieved using a relatively simple marking algorithm. The other three systems have used more complex marking algorithms but the human-system agreement rates are similar to those of IndusMarker.

10 Conclusion

The salient feature of IndusMarker (from the pedagogical perspective) is that it can provide practice tests and immediate feedback to students regardless of the size of the class. The lecturer has to initially spend some time developing and validating the required structures when a practice test is conducted for the first time. But, once the required structures are finalized, the same practice test may be repeated wherever the same course material is taught. The lecturer conducting the practice test, after the first test, does not need to spend any time manually marking student's tests. This paper has described the use of the system in the context of an OOP course at BU. The authors have received positive feedback about the system from the lecturers involved and also from students of the OOP course. IndusMarker can be further enhanced by including features that can provide detailed statistical analysis of students' performances for both lecturers and students so that each may adjust or modify their teaching or learning approach for the course.

In recent years, a number of automated marking systems for program texts, i.e., texts written in a programming language, have been developed [28]. However, these systems cannot mark short answers expressed in natural language. Short-answer questions provide a very useful means of testing theoretical concepts associated with a programming course. This is the first time that a short-answer marking system has been used to provide automatic summative feedback to students and teachers about students' understanding of theoretical concepts in a programming course. IndusMarker may be integrated with other program marking systems to form a single system that can mark both programming exercises as well as short-answer questions.

References

[1] K.E. Holbert and G.G. Karady, "Strategies, Challenges and Prospects for Active Learning in the Computer-Based Classroom," *IEEE Trans. Education,* vol. 52, no. 1, pp. 31-38, Feb. 2009.
[2] C. Leacock and M. Chodorow, "C-Rater: Automated Scoring of Short-Answer Question," *Computers and the Humanities,* vol. 37, no. 4, pp. 389-405, 2003.
[3] J.Z. Sukkarieh and S.G. Pulman, "Automatic Short Answer Marking," *Proc. Second Workshop Building Educational Applications Using NLP,* pp. 9-16, June 2005.
[4] T. Mitchell, T. Russel, P. Broomhead, and N. Aldridge, "Towards Robust Computerized Marking of Free-Text Responses," *Proc. Sixth Int'l Computer Assisted Assessment Conf.,* 2002.
[5] R. Siddiqi and C.J. Harrison, "A Systematic Approach to the Automated Marking of Short-Answer Questions," *Proc. 12th IEEE Int'l Multi Topic Conf. (IEEE INMIC),* pp. 329-332, 2008.
[6] M. Farrow and P.J.B. King, "Experiences with Online Programming Examinations," *IEEE Trans. Education,* vol. 51, no. 2, pp. 251-255, May 2008.
[7] Course Website of the "Object Oriented Programming" Course, http://sites.google.com/site/sen142oop/Home, 2009.
[8] Official Website of Bahria Univ., http://bimcs.edu.pk/about-bahria/why-bahria.htm, 2009.
[9] Official Website of Higher Education Commission Pakistan, http://www.hec.gov.pk, 2010.
[10] Official Website of Pakistan Eng. Council, http://www.pec.org.pk, 2010.
[11] J.H. McMillan, *Classroom Assessment: Principles and Practice for Effective Instruction,* pp. 40 and 117. Allyn and Bacon, 1997.
[12] J.T. Dillon, *Questioning and Teaching: A Manual of Practice.* Teachers College Press, 1988.
[13] L.R. Gay, "The Comparative Effects of Multiple-Choice versus Short-Answer Tests on Retention," *J. Educational Measurement,* vol. 17, no. 1, pp. 45-50, Spring 1980.
[14] M.A. McDaniel, J.L. Anderson, M.H. Derbish, and N. Morrisette, "Testing the Testing Effect in the Classroom," *European J. Cognitive Psychology,* vol. 19, nos. 4/5, pp. 494-513, 2007.
[15] W.J. Hanson, "Creation of Hierarchic Text with a Computer Display," PhD thesis, Stanford Univ., 1971.
[16] S. Holzner, *Inside XML,* pp. 9-10. New Riders, 2001.
[17] D. Gulbransen, *The Complete Idiot's Guide to XML,* pp. 131-133. Que, 2000.
[18] S.S. Laurent and E. Cerami, *Building XML Applications,* p. 46. McGraw-Hill, 1999.
[19] The Stanford Natural Language Processing Group, "The Stanford Parser: A Statistical Parser," http://nlp.stanford.edu/software/lex-parser.shtml, 2009.
[20] i-net Software, "JOrtho—A Java Spell-Checking Library," http://www.inetsoftware.de/other-products/jortho, 2009.
[21] Wiktionary, "Wiktionary—A Wiki-Based Open Content Dictionary," http://wiktionary.org, 2009.
[22] M.C. de Marneffe, B. MacCartney, and C.D. Manning, "Generating Typed Dependency Parses from Phrase Structure Parses," *Proc. Fifth Int'l Conf. Language Resources and Evaluation,* pp. 449-454, 2006.
[23] B. McLaughlin, *Java and XML,* pp. 125-135. O'Reilly & Assoc., 2000.
[24] K. Cagle, M. Corning, J. Diamond, T. Duynstee, O.G. Gudmundsson, M. Mason, J. Pinnock, P. Spencer, J. Tang, and A. Watt, *Professional XSL.* Wrox Press, 2001.
[25] F. Marton and R. Saljo, "On Qualitative Differences in Learning—I: Outcome and Process," *British J. Educational Psychology,* vol. 46, pp. 4-11, 1976.
[26] L.S. Ming, "An ICT-Based Technique for Assessment of Short-Answer Question," *Proc. Seventh Int'l Conf. Information Technology Based Higher Education and Training,* 2006.
[27] R. Siddiqi and C.J. Harrison, "On the Automated Assessment of Short Free-Text Responses," *Proc. 34th Ann. Conf. Int'l Assoc. for Educational Assessment (IAEA),* 2008.
[28] C. Daly and J.M. Horgan, "An Automated Learning System for Java Programming," *IEEE Trans. Education,* vol. 47, no. 1, pp. 10-17, Feb. 2004.

Raheel Siddiqi received the MSc degree in information systems engineering from the University of Manchester Institute of Science and Technology (UMIST), United Kingdom, in 2004. He then worked as a lecturer at the Sir Syed University of Engineering and Technology (SSUET), Karachi, Pakistan, for two years. He has also taught as a visiting faculty member at the Institute of Business Administration (IBA), Karachi, Pakistan. Currently, he is pursuing the PhD degree from The University of Manchester, United Kingdom. The title of his research is "Automated Short-Answer Marking through Text Structure Analysis."

Christopher J. Harrison received the MSc (Thesis) and PhD degrees from the University of Manchester Institute of Science and Technology (UMIST), United Kingdom, in 1986 and 1999, respectively. He joined the Department of Computation at UMIST as a lecturer in 1988. On the creation of The University of Manchester, October 22, 2004, he became a lecturer in the School of Informatics, The University of Manchester. He is currently a lecturer in the School of Computer Science, The University of Manchester. His research interests include persistent object stores, software tools, programming and software specification languages, and integrated computer-based teaching systems.

Rosheena Siddiqi received the MSc degree in information systems engineering from the University of Manchester Institute of Science and Technology (UMIST), United Kingdom, in 2004. She has been teaching computer programming at various universities for the last four years. She is currently a lecturer at Bahria University, Pakistan. Her research interests include computer-aided assessment and programming languages.