Computing and Informatics, Vol. 27, 2008, 681–698

XML-BASED AUTOMATIC TEST DATA GENERATION

Halil Ibrahim Bulbul

Department of Computer Education
Gazi University, Ankara, Turkey
e-mail: `bhalil@gazi.edu.tr`

Turgut Bakir

Siemens AG, Ankara, Turkey
e-mail: `turgut.bakir@siemens.com`

Manuscript received 31 October 2006; revised 6 August 2007
Communicated by Mária Bieliková

Abstract. Software engineering aims at increasing quality and reliability while decreasing the cost of the software. Testing is one of the most time-consuming phases of the software development lifecycle. Improvement in software testing results in decrease in cost and increase in quality of the software. Automation in software testing is one of the most popular ways of software cost reduction and reliability improvement. In our work we propose a system called XML-based automatic test data generation that generates the test data automatically according to the given data definition. We also proposed a test data definition language to describe the test data to be generated. This system reduces the testing time compared to manual test data generation and increases the testing reliability compared to the random test data generation by eliminating meaningless test data.

Keywords: XML, XSL, XSLT, test data generation

1 INTRODUCTION

Software testing is a critical element of a software quality assurance that aims at determining the quality of the system and its related models. In such a process, a software system will be executed to determine whether it matches its specification and executes in its intended environment [1]. Software testing is an expensive and time-consuming process, especially in safety-critical applications, typically consuming at least 50 % of the total cost involved in developing software [2]. Thus, decreasing the sofrware testing costs decreases the overall development cost significantly. In order to reduce the high cost of manual software testing and at the same time to increase the reliability of the testing processes researchers and practitioners have tried to automate it. One of the most important components in a testing environment is an automatic test data generator – a system that automatically generates test data for a given program [3]. There are a lot of works done most of which mainly aim at automating the testing process.

Testing can be automates in different ways to minimize the manual tasks. Software testing automation processes include, but are not limited to, automatic test cases, automatic running or simulation, or automatic test data generation. Since manual test data generation is one of the most expensive parts of testing, in our work we focused on automatic data generation process.

There are many works referring to automatic test data generation in the literature. Some of them are focused on specific type of data such as floating-point numbers or integers [4, 5], arrays or pointers [6]. Most of them are developed for specific software such as Ada [7], ML [8] or Java [9]; other works propose new algorithms for test data generation. Genetic algorithms [7, 10, 11], or dynamic domain reduction algorithm [12] can be given as examples of proposed test data generation algorithms.

The above works are pretty good examples of the progress in test data generation. Each of them has its own advantages and disadvantages compared to the others. However, in our work we did not focus on a new algorithm or specific software as listed above. Our main objective is to achieve a generic system which automatically generates data for any software, which accepts inputs of any type. In order to achieve this goal, properties of the data to be generated should be clearly defined. Maurer [13] proposed a context-free grammar-based language for generating test data. In this method a context-free grammar-based language called DGL is developed. In Maurer's work data is defined syntactically as a string of characters. This definition is then given to a compiler which generates C code accordingly. Although this is a pretty good way to generate the test data, it has some disadvantages. First, as mentioned in [13], the system has quite complex parts and the compiler has some design problems. Another disadvantage is the utilization of a compiler, which limits the produced code. The coding part of this work seems to be static because of its complex structure. In our work we tried to develop a system which is more dynamic for new algorithms.

Under these considerations we tried to find out an easier and flexible method. In our work we utilized XML (eXtensible Markup Language) [15] technology which is very recent and popular. XML is a meta language, which is a subset of SGML (Standard Generalized Markup Language). Today XML has a very wide range of usage mainly in internet technologies. An XML stream can be used as the definition of any type of structured data. The XML stream format can be defined by DTD (Document Type Declaration) or XSL (XML Schema Language) [16]. As the first step of our work we defined the rules for test data definition language. This work is very similar to that done in ATLAS test environment (ATE) [17]. Instead of writing a parser and a compiler for the definition given in XML stream we defined transformation rules which are independent of programming language and environment. We used XSLT (XML Schema Language Transformations) [18] technology which defines transformation rules of an XML stream input. XSLT can be defined so that an XML stream can be transformed to any type of output such as a text or an HTML document. Unlike ATE our system can be used to generate a code in any programming language. To do this, in our system changing the XSLT rules is enough.

In Figure 1 a general scheme of the overall system is given. As seen from the figure, test data definition language rules and language transformation rules are defined. In practical usage of the system, test data is defined according to the language rules. After validation of the definition given, the transformation rules are then applied to the definition by utilizing a transformer factory. This results in a code which is called "Test Data Generator" dynamically. The resulting code is not necessarily in Pascal or C language. If its transformation rules are defined in XSLT then the resulting code may be generated in any programming language.

In this article we introduced our system, including a test data definition part, and automatic test data generation part. In Section 2, we explained the test data definition language rules. In Section 3 we introduced the automatic test data generation rules defined by XML transformation rules. Then we depict some simple examples showing the usage of this system. We conclude this article by summarizing the advantages and disadvantages of the developed system (Section 7).

2 TEST DATA DEFINITION LANGUAGE

Test data generators can be classified into three types: pathwise test data generators, data specification systems, and random test data generators [19]. In pathwise test data generators the inputs are selected to exercise a specified set of program paths. A data specification system assists users in generating a test case by providing a data specification language to describe the input data. In random test generators test data is generated by simply selecting a random point from the domain of each input variable of a program.

In our work we realized a system for automatic test data generation classified in data specification systems. Such systems aim at providing feasible test data in software testing. Since random input is of little value in verifying software correctness, the controlled way of test data generation should be preferred. Software testing with randomly generated data lasts longer than other methods for verification of correctness of software, since more input is required. Narrowing the input domain and limiting the input with some specification removes infeasible data

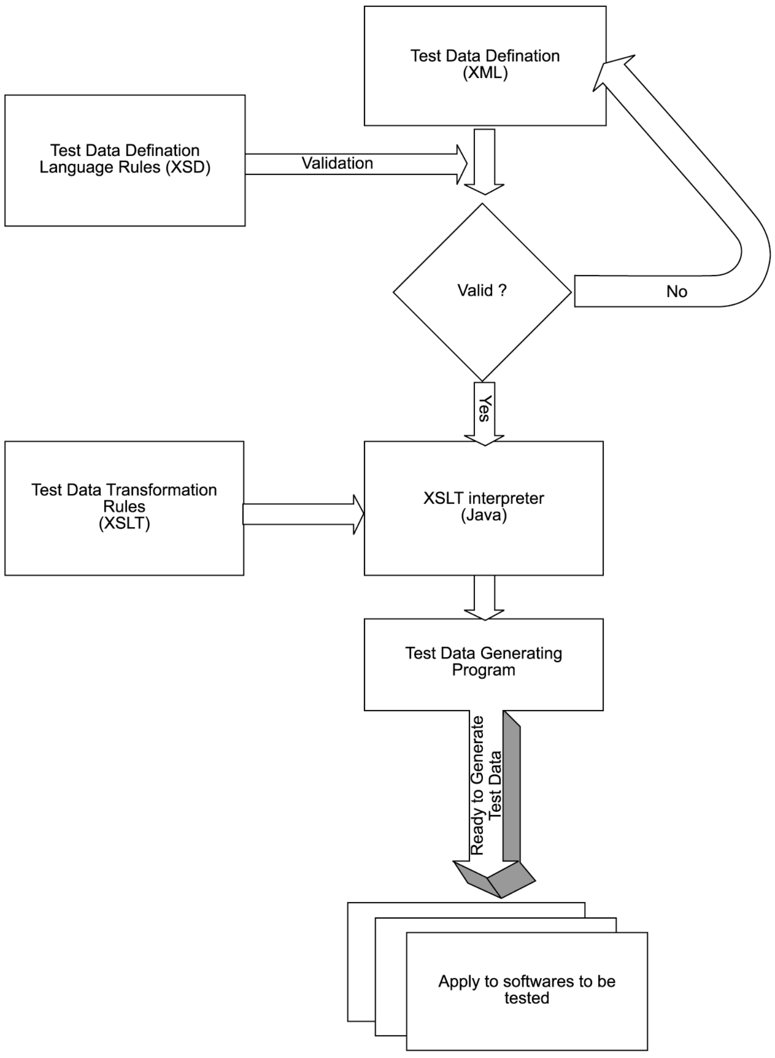

Test Data Defination (XML)
Test Data Defination Language Rules (XSD)
Validation
Valid ?
No
Yes
Test Data Transformation Rules (XSLT)
XSLT interpreter (Java)
Test Data Generating Program
Ready to Generate Test Data
Apply to softwares to be tested

Fig. 1. Workflow diagram of overall structure

from test data set. It saved time in testing phase by passing meaningless testing cases.

In this area Maurer [13] proposed a system utilizing context-free grammars. Unlike this work we utilized XML technology for definition of test data. By utilizing XML in this work we took very well known advantages of this technology. Although originally designed for large-scale electronic publishing, XML has found a role in supporting the exchange of a wide variety of data on the web and in other software systems [15]. Its flexibility made our system flexible as well. Since it is extensible, our system is also extensible. The following explanations will give more details about the reasons for selecting XML as the basis for our definition language.

In order to propose a language, grammatical rules should be clearly identified. For this purpose we start with identification of language rules. Unlike ATE [17] we chose XML schema definition (XSD) instead of data type definition (DTD) language. XSD is a newer schema language than DTD. Detailed information on XML schema and advantages over DTD is found in [20].

2.1 Language Components

Our test data definition language has two main components – the domain part and the output part. The domain part is the definition of the domain from which the generated test data will be selected. The output part contains the behaviors/rules of the test data selection such as exclusions, preferences and methods.

The element "data_domain" has child elements giving detailed description of the domain. We defined the domain type element to classify the data type of the domain. Up to now in language definition rules we restricted the domain type with three data types, namely "String", "Integer" and "Decimal" . It is open for new data type definitions like the other parts of language rules. The domain itself can be a "Set" or a "Range". The domain can also be a file which contains the elements of the domain. In each case, the data domain is a set of values. This definition only gives flexibility and convenience in the domain set definition.

Although giving domain definition by "Set" element is a fatiguing way of domain definition, it has advantages when defining a small set of data. For example, if your software accepts currency input like 'USD', 'EURO' and 'TRL', it is easy to define this domain with only three elements. On the other hand, if your domain is large unlike the currency example, it is not feasible to use "Set" definition. In this case "Range" type of domain definition should be utilized if possible. For example, if our software accepts input "Amount" data of type decimal we can define a range from a negative value to a very big positive value. If we want to select amounts with negative values and very large values only, then we may include them in a file. Higher level definition rule of "data_element" is given in Figure 2.

Definition of the data domain without any selection specification is nothing but definition of random data generation. Our domain definition part gives a little more restrictions than random data generation. In some cases this is not enough; then, test data properties or selection methods should be defined as well. The output part

```
<xsd:element name="data_domain">
 <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="domain_type" type="data_type" minOccurs="1"
      maxOccurs="1"/>
      <xsd:element name="values">
         <xsd:complexType>
             <xsd:choice>
               <xsd:element name="domain_Set" type="Set" minOccurs="1"
                 maxOccurs="1"/>
               <xsd:element ref="Range" minOccurs="1" maxOccurs="1"/>
               <xsd:element ref="file" minOccurs="1" maxOccurs="1"/>
              </xsd:choice>
         </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Fig. 2. "Domain_type" definition rules

of the test data definition defines the test data selection rules. The rules defined in this part are also extensible for new requirements in test data generation.

The second part of the test data definition is the "output" element part in which the test data generator's behavior is defined. The child elements and properties of this part guide the test data selection methods from the domain given in the first part. The very basic element of the output element is the "number_of_element" part which specifies the number of test data to be generated. According to this value the loop counter in the test data generator is declared. The "preffered_output_set" and the "exclusion_set" elements identify which values are wanted to be generated in the test data and which values are not. It is useful to define the domain as a range and to exclude some elements from it. For example, if "0" value is meaningless for our software we can exclude it to avoid unnecessary test executions.

The "mean" and "variance" elements are defined for statistical approaches. In the generated test data set a mean or a variance value is to be reached. For the statistical ways of automatic test data generation algorithms the language can be extended by adding new attribute or element rules into the language.

The "Successive_data" element is defined mainly for string type test data. If different successive data input is required then the value must be set to false, otherwise true. This is an example of a grammatical approach for automatic test data generation.

In order to include a specific test data generation algorithm, the "method" element is included into the test data definition language. The value of this element is the name of the algorithm to be utilized. The algorithms as given in the introductory part and others which have already been developed are very valuable in their proposed area. Thus, development of such methods from the beginning is nothing but waste of time. If we have library of such valuable methods in our test data generation part, we can take advantage of already developed and used systems.

The core language rules for output element are given in Figure 3.

```
<xsd:element name="output">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="number_of_output" type="xsd:integer"
        maxOccurs="1"/>
      <xsd:element name="method" type="xsd:string" />
      <xsd:element name="preffered_output_set" type="Set" minOccurs="0"
        maxOccurs="1"/>
      <xsd:element name="exclusion_set" type="Set" minOccurs="0"
        maxOccurs="1"/>
      <xsd:element ref="mean" minOccurs="0" maxOccurs="1"/>
      <xsd:element name="variance" type="xsd:decimal"/>
      <xsd:element ref="output_type" minOccurs="1" maxOccurs="1"/>
      <xsd:element ref="file" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
    <xsd:attributeGroup ref="Successive_data"/>
  </xsd:complexType>
</xsd:element>
<xsd:attributeGroup name="Successive_data">
  <xsd:attribute name="same_successive_data" type="xsd:boolean"
    use="optional"/>
  <xsd:attribute name="same_saccessive_data_occ" type="xsd:decimal"
    use="optional"/>
</xsd:attributeGroup>
```

Fig. 3. Output definition rules (continued)

The last part of our language is the "generate_what" element, defined in Figure 4. In this part the form of the output itself is defined. There are several choices. The resulting output may be a program which is a test data generator. The result may be a function, procedure or a unit (or all). In this case the output is a program which can be used in other programs. It can be included in the program to be tested as well. The main goal of this part which is not implemented in this work, is supplying input data for a testing software which automatically executes the program to be tested. For such automatic testing software the input would be supplied by this function or procedure output of out test data generation system. In our work, we used existing software testing tools such as CATT [21] tool of the SAP [22] system. This system executes the program by using external input in specified file format (either binary or text). In our test we generated the test data generator program and gave its output to SAP's CATT tool as input.

```
<xsd:simpleType name="generate_what">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Procedure"/>
    <xsd:enumeration value="Function"/>
    <xsd:enumeration value="Program"/>
    <xsd:enumeration value="Unit"/>
    <xsd:enumeration value="Data_Stream"/>
  </xsd:restriction>
</xsd:simpleType>
```

Fig. 4. "Generate_what" element definition rules

2.2 XSLT Interpreter – Generating Output

XML is extensible, easy to write and understand. Two main features named "valid" and "well-formed" guarantee that the input stream is valid and well-formed.

In our project the test data definition is written in XML format, but it does not mean that any XML stream which is well-formed is a definition of test data. A definition should obey the rules defined in XSD file. Each XML file must start with a line, which tells the parser that the rules of our language are defined somewhere. The meaning of this expression is "I obey the language rules defined in this XSD, otherwise do not accept me" . An example is given in Figure 5.

```
<test_Data  xmlns="http://localhost"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=http://localhost TestDataGenerationSchema.xsd >
```

Fig. 5. Root element "test_data" definition

Before processing the test data definition we must be sure that the given definition is a valid test data definition. To check the validity of the XML input packages developed for most of the popular programming languages and environments can be used. This is one of the main reasons which made us to select this technology.

Little effort is required to write a code for XML validity assurance. In our work we used Java which is the most popular programming language today. In Figure 5, we gave the full code of the "parse" method which gets the XML file as input and parses it. If XML input is a valid test data definition then the parser accepts it, otherwise it fails.

```
public Document parse(String xmlFileURL) {
  try {
    DocumentBuilderFactory docFactory =
      DocumentBuilderFactory.newInstance();
    docFactory.setValidating(true);
    docFactory.setNamespaceAware(true);
    DocumentBuilder builder = docFactory.newDocumentBuilder();
    MyErrorHandler eh = new MyErrorHandler(System.out);
    builder.setErrorHandler(eh);
    Document document = builder.parse(xmlFileURL);
    System.out.println(document.toString());
    return document;
  } catch(Exception ex)
  {
    ex.printStackTrace();
    return null;
   }
}
```

Fig. 6. XML validation code written in Java

2.3 An Example

In order to illustrate a test data definition an example is given in Figure 7. It tells that we want to produce sixty-integer output from an integer range from 1 to 5 000. In the output stream, we prefer to have "660, 350, 400, 330, 750, 200" more often. The mean value of the test data values will approximately be 600. Finally, we say that we want to have a resulting program file named "TestData.pas" which will produce this test data stream.

We can summarize the main advantages of the test data definition based on XML technology as follows:

- It is easy to understand and to write. Since XML has very basic rules, it is easy to write and understand.
- It has a wide application area. In this work there is no specific target software or data domain. The rules are so generalized that test data for any software system can be defined.
- It is extensible for new needs. As a result of the nature of XML, our language is an extensible structure. Schema definition is open for new rules and extensions.
- It is environment-independent. Our language is not platform-dependent. That is, it is supported by any XML parser developed in any programming language, and in any operating system.

```
<?xml version="1.0" encoding="UTF-8"?>
<test_Data xmlns="http://localhost"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://localhost TestDataGenerationSchema.xsd ">
  <data_domain>
    <domain_type>Integer</domain_type>
    <values>
      <Range>
        <Integer_range>
          <Lower_bound>1</Lower_bound>
          <Upper_bound>5000</Upper_bound>
        </Integer_range>
      </Range>
    </values>
  </data_domain>
  <output>
    <number_of_output>60</number_of_output>
    <preffered_output_set>
      <String_set>
        <integer_element>660</integer_element>
        <integer_element>350</integer_element>
        <integer_element>400</integer_element>
        <integer_element>330</integer_element>
        <integer_element>750</integer_element>
        <integer_element>200</integer_element>
      </String_set>
    </preffered_output_set>
    <mean>600</mean>
    <output_type>
      <generate_what>Program</generate_what>
    </output_type>
    <file>c:/program/TestData.pas</file>
  </output>
</test_Data>
```

Fig. 7. A simple test data definition example

3 TEST DATA GENERATION

In this work, instead of generating test data directly we create a program dynamically to generate test data. The resulting program is called test data generator. The advantages are as follows:

- flexible,
- easy to maintain,

- multi programming language can be supported,
- the resulting test data generator can be included by other programs.

There are some alternatives to generate the test data generator as explained below.

One alternative is to write a program which takes the test data definition and produces the test data directly. It is not the best choice since it is hard to maintain. It would be a platform dependent solution. Supported data types can not be extended easily.

The other alternative is to write a compiler-like program which takes the test data definition and produces the test data generator. This would be a better solution than that mentioned above; but this is also hard to extend for the new requirements. Which this solution, the programming language of the test data generator can not be changed.

The solution that we decided to follow is using XSLT technology which is platform- and language-independent solution. This gave us flexibility to extend the supported data types and properties of the test data to be generated. As explained below it is easy to update for the new requirements.

4 PARSING AND TRANSFORMING XML

In our project we utilized standard XML functions to create test data generator dynamically. XSLT is the best choice we found for our process. Unlike Atlas test environment [17], instead of writing a compiler we created an XSLT file which contains transformation rules. These rules can be updated via an ordinary editor to change or add new functions. This work does not depend on any programming language which is the main advantage of the utilization of XML schema transformation over the other techniques.

The work flow given in Figure 8 shows how transformation is processed. Starting at the root element the XML is parsed and the output is produced according to the next sibling elements. For example, if the program finds "program" in the generate_what element value, a "PROGRAM" statement is written to the output. Then the parser searches for the Data_domain elements for constant and variable declarations.

After building the declaration part of the target source code, the program body is built according to the output complex element. As explained in the Test Data Definition section of this article, output element is a complex element which gives keys to create the program body. The number_of_output, for example, defines the count of the "for" loop. As another example, according to the exclusion_set, in target source code an exclusion set is defined and data produced in the "for" loop is tested whether it is in the exclusion set or not. Thus, an appropriate "if statement" is inserted into the target program for the exclusion set.

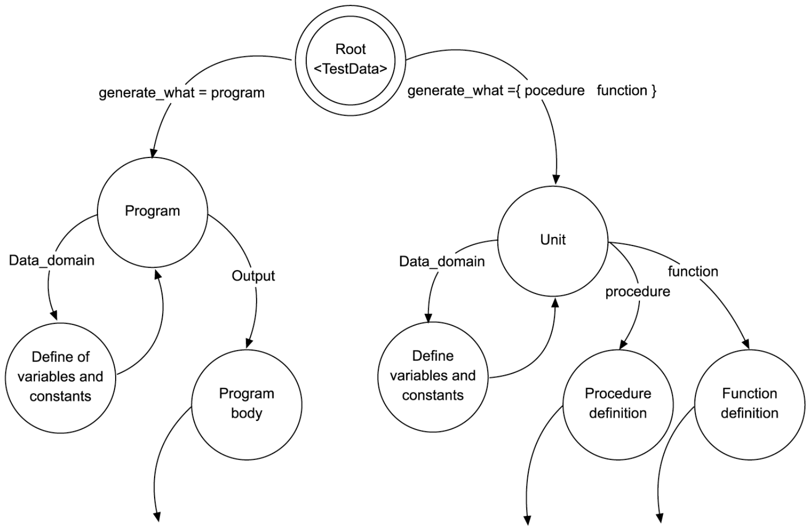


Fig. 8. Workflow diagram for XSLT transformation

5 SOURCE CODE GENERATION

As explained above, the source code generation rules are defined in XSLT file. As seen from Figure 1, XSLT file is applied to the XML file which is the test data definition. It results in a source code named "Test Data Generator" . Test data generator is a program created according to the transformation rules given. We wrote a small Java program which gets XML file and XSLT file as input and test data generator as output. This code is given in Figure 9.

In our work we have chosen Pascal and C programming languages for the generated source code. This does not mean that our test data generating system can generate source code in these programming languages only. We have shown that by changing XML transformation rules we easily change the test data generator programming language; no additional change is necessary to do that. This gives us language and environment independency. At any time the source code generation can be changed by changing the XML transformation rules.

As an example, when the parser finds "Program" value in the generate_what element, the "Program" statement is inserted into the target source code if the Pascal code is created. The same value causes "public void main ()" statement in the target source code if the language is C. For other programming languages we can make similar changes in XSLT file to generate the source code.

```
package com.gazi.project;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
public class Transform {
  public Transform() { super(); }
  public static void main(String[] args) {
    String xslFile = "";
    String xmlFile = "";
    String outFile = "";
    for(int i=0; i < args.length ; i++) {
      if(args[i].equalsIgnoreCase("XSLFile"))
      xslFile = args[++i] ;
      if(args[i].equalsIgnoreCase("XMLFile"))
      xmlFile = args[++i] ;
    }
    GaziParserDom domParser = new GaziParserDom();
    outFile = domParser.search(xmlFile,"file");
    TransformerFactory factory = TransformerFactory.newInstance();
    try {
      System.out.println("Transforming Information...");
      Transformer transformer_info =
      factory.newTransformer(new StreamSource(xslFile));
      transformer_info.transform(new StreamSource(xmlFile),
        new StreamResult(outFile));
      System.out.println("Automatic test data genration "+
        "program completed.");
      System.out.println(outFile + "Created");
    } catch (TransformerException e) { e.printStackTrace(); }
  }
}
```

Fig. 9. The Java code that parses XML and creates the test data generator

6 EXPERIMENTAL WORK

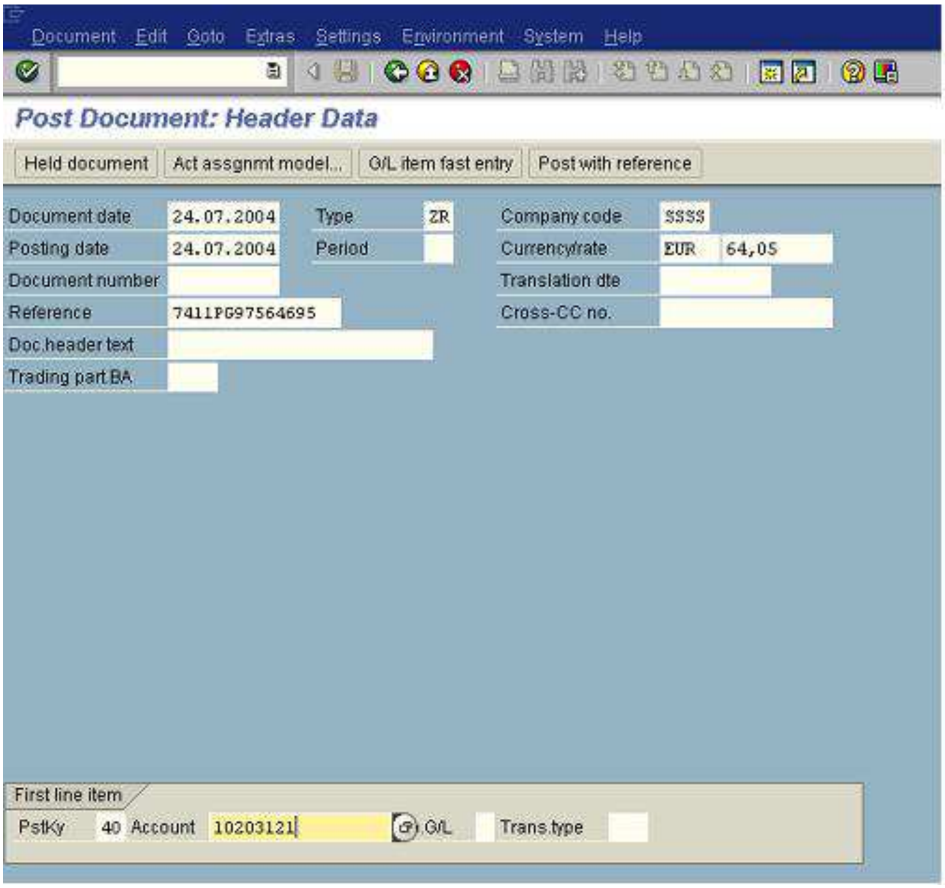
In this work we made experimental work for different use cases. Here we give a simple example that shows how this system is used in real life. The given problem is to test an SAP (The most popular ERP software) [22] module which creates the accountancy record. The required data to test this module include the reference number, amount, currency and cost center. The reference number should be a string in the range of "7411RF00000000" and "7411RF99999999" . The amount is a decimal number between 10 and 999999999.99. Currency should be selected from

```
<?xml version="1.0" encoding="UTF-8"?>
<test_Data xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://www.w3.org/2001/XMLSchema TestDataGenerationSchema.xsd">
  <data_domain>
    <domain_type>String</domain_type>
    <values>  <Range>  <String_range>
          <Lower_bound>7411RF00000000</Lower_bound>
          <Upper_bound>7411RF99999999</Upper_bound>
    </String_range>  </Range>  </values>
    <domain_type>Decimal</domain_type>
    <values>  <Range>  <Decimal_range>
          <Lower_bound>10.00</Lower_bound>
          <Upper_bound>999999999.99</Upper_bound>
       </Decimal_range>  </Range>  </values>
    <domain_type>String</domain_type>
    <values>  <domain_Set>  <String_set>
      <string_element>TRL</string_element>
      <string_element>EUR</string_element>
      <string_element>USD</string_element>
    </String_set>  </domain_Set>  </values>
    <domain_type>String</domain_type>
    <values>  <domain_Set>  <String_set>
        <string_element>C151</string_element>
        <string_element>C152</string_element>
        <string_element>C153</string_element>
        <string_element>A110</string_element>
        <string_element>C160</string_element>
        <string_element>F512</string_element>
    </String_set>  </domain_Set>  </values>
  </data_domain>
  <output same_successive_data="true">
    <number_of_output>8</number_of_output>
    <method>random_string</method>
    <output_type name="GenerateTestData">
      <generate_what>Program</generate_what>
    </output_type>
    <file>D:/WSAD_WS/Test Data Generation/Test_XMLs/testdata.pas</file>
  </output>
</test_Data>
```

Fig. 10. Test data definition as an example

```
7411PG97564695 64,05 EUR C153
7411PG97564692 315,32 EUR F512
7411PG97564693 164,46 EUR C160
7411PG97564688 133,4 EUR C153
7411PG97594596 8,8 EUR C151
7411PG97594597 2.257,2 EUR F512
7411PG97594595 141,49 EUR C151
7411PG97594585 228 EUR C151
```

Fig. 11. Output of the test data generator



Post Document: Header Data

Document date	24.07.2004	Type	ZR	Company code	SSSS
Posting date	24.07.2004	Period		Currency/rate	EUR 64,05
Document number				Translation dte	
Reference	7411PG97564695			Cross-CC no.	
Doc.header text					
Trading part.BA					

First line item

PstKy 40 Account 10203121 G/L Trans.type

Fig. 12. Application of the generated test data to the module to be tested

the set of strings which is equal to the set {"TRL" , "EUR" , "USD" }. Finally, the cost center is selected from the string set {"C151", "C152", "C153", "A110", "C160", "F512"}.

We have a module in SAP which reads all required data from a text file and executes the accountancy record creating the module. Our test data generator which is created automatically from the definition of the test data creates a text file which consists of an automatically generated data set. The output of this generator are given as input to the module to be tested.

Test data definition, generated test data, and applied module are given in Figures 10–12.

7 CONCLUSION AND FUTURE WORK

In this article we presented automatic test data generation based on the XML test data definition. We defined an XML-based language called "Test Data Definition Language" which identifies the test data to be produced. We also defined a set of transformation rules used in source code generation. In this work not all the properties that would be needed are defined. Since there is no hard coded program, it is open for development. If there are new needs they will be identified easily. Test data definition language can be developed by new requirements. Similarly to test data definition language, transformation rules are also open for development. Since XSLT does not provide functions for date type values, additional work should be performed to handle test date in date type. In this work we tried to define transformation rules for Pascal and C languages. As we explained in this article it can be any programming language.

REFERENCES

[1] Keyes, J.: Software Engineering Handbook. CRC Press LLC, 2003.

[2] Beizer, B.: Software Testing Techniques. Thomson Computer Press, 2nd edition, 1990.

[3] Edvardsoson, J.: A Survey on Automatic Test Data Generation. Proceedings of the Second Conference on Computer Science and Engineering in Linköping, 1999, Vol. 2128.

[4] Transy, N.—Deville, Y.: Automatic Test Data Generation for Programs with Integer and Float Variables. Proceedings of the Automated Software Engineering Conference, IEEE Computer Society, San Diego, 2001, pp. 13–21.

[5] Miller, W.—Spooner, D.: Automatic Generation of Floating-Point Test Data. IEEE Transactions on Software Engineering, SE-2, Vol. 3: September 1976, pp. 233–226.

[6] Korel, B.: Automated Software Test Data Generation. IEEE Transactions on Software Engineering, Vol. 16, 1990, No. 8, pp. 870–879.

[7] Jones, B. F.—Sthamer, H. H.—Eyres, D. E.: Generating Test-Data for Ada Procedures Using Genetic Algorithms. In Genetic Algorithms in Engineering Systems: Innovations and Applications. IEEE, September 1995, pp. 65–70.

[8] Ryu, S.—Yi, K.: Automatic Test Data Generation for Exceptions in First-Order ML Programs. Research On Program Analysis System, ROPAS Memo 1999-3, November 1999.

[9] Boyapati, C.—Khurshid, S.—Marinov, D.: Korat: Automated Testing Based on Java Predicates. ACM International Symposium on Software Testing and Analysis (ISSTA), July 2002.

[10] Watkins, A. L.: The Automatic Generation of Test Data Using Genetic Algorithms. Proceedings of the 4th Software Quality Conference, Vol. 2, 1995, pp. 300–309.

[11] Pargas, R.—Harrold, M. J.—Peck, R.: Test-Data Generation Using Genetic Algorithms. Journal of Software Testing, Verifications, and Reliability, Vol. 9, 1999, pp. 263–282.

[12] Offutt, A. J.—Jin, Z.—Pan, J.: The Dynamic Domain Reduction Procedure for Test Data Generation. Department of Information and Software Engineering George Mason University Technical Reports, 1994, ISSE-TR-94, pp. 94–110.

[13] Maurer, P. M.: The Design and Implementation of a Grammar-based Data Generator. IEEE Software-Practice and Experience. Vol. 22, 1992, No. 3, pp. 223–244.

[14] Maurer, P. M.: Generating Test Data with Enhanced Context Free Grammars. IEEE Software, Vol. 7, 1990, No. 4, pp. 50–56.

[15] Bray, T.—Paoli, J.—Sperberg, C. M.: Extensible Markup Language (XML) 1.0. W3C Recommendation, 10 February 1998. `http://www.w3.org/TR/1998/REC-xml-19980210`.

[16] Berglund, A.: Extensible Stylesheet Language (XSL). Version 1.1, W3C Working Draft, 17 December 2003. `http://www.w3.org/TR/2003/WD-xsl11-20031217/`.

[17] Chen, C.—Lee, J. K.: Case Study: An Infrastructure for C/ATLAS Environment with Object-Oriented Design and XML Representation. Journal of System and Software, Vol. 71, 2004, No. 1–2, pp. 83–95.

[18] Clark, J.: XSL Transformations (XSLT). Version 1.0, W3C Recommendation, November 16, 1999, `http://www.w3.org/TR/xslt/`.

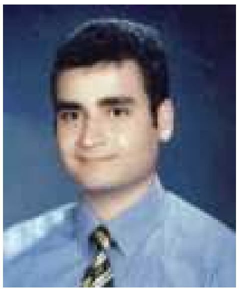
[19] DeMillo, I.: A Richard, Software Testing and Evaluation. The Benjamin/Cummings Publishing Company, Inc., 1987, pp. 106–115.

[20] Fallside, D. C.—Walmsley, P.: XML Schema Part 0: Primer Second Edition. W3C Recommendation, 28 October 2004, `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`.

[21] `http://www.sap.com/services/servsuptech/brochures/index.aspx`.

[22] `http://www.sap.com`.

Halil Ibrahim Bulbul received his Ph. D. degree in educational technology from Ankara University, Ankara, Turkey, and his M. Sc. degree in technology education from California University (PA), U. S. A., in 1997 and 1990, respectively, and his B. Sc. degree in technology education from Gazi University, Ankara, Turkey, in 1985. He has been an Assistant Professor of Computer Education Department, Gazi University, Ankara, Turkey, since 1997. His research interests include educational technologies, e-learning, distance education, educational software design, and database management systems.



Turgut Bakir received B. Sc. degree in computer engineering from Middle East Technical University, Ankara, Turkey, and his M. Sc. degree in computer education from Gazi University Ankara, Turkey, in 2005. He is currently working for Siemens Turkey as SAP specialist and software engineer. His research interests include automated software testing, dynamic program generation, and ERP systems.