Lean Implementations of Software Testing Tools Using XML Representations of Source Codes

Yu Xia Sun *+ Huo Yan Chen +
Department of Computer Science
* Sun Yat-sen University; + Jinan University
Guangzhou, P.R. China
jnusyx@office.jnu.edu.cn; tchy@jnu.edu.cn

T.H. Tse §
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

***Abstract*—By utilizing XML representations of source programs under test, a new approach is proposed to concisely implement some prototypes for TACCLE, a software testing methodology. The conversions between a source program and its XML representation can be easily realized using existing conversion tools. In this way, the conversion tools can automatically analyze and parse the source program, so that testing tool developers only need to concentrate on the manipulation of the XML document. If appropriate XML DOM APIs are chosen, the implementations of such testing tools will be pretty lean. A detailed case study for GMPS tool, a prototype for the TACCLE methodology, is presented to illustrate the new approach.**

Keywords—software testing; lean implementation; XML representation; conversion tool

I. INTRODUCTION

When developing different software testing tools, the process of analyzing and parsing the source Program Under Test (referred to as *PUT*) is repetitive. For instance, when we implemented a series of prototype tools (such as DOE [9] and GMPS [8]) for the software testing methodology TACCLE [4], we also repeated that process. In our previous approaches, we adopted compiling and interpreting techniques for implementation. Although we utilized the object-oriented paradigm to achieve reuse in analysis and parsing, the implementations of these tools were still complex and bulky. Also, the codes for parsing greatly exceeded those for manipulating the source *PUT*.

Actually, program parsing and analysis are widely used when developing white-box testing tools. Hence, it is significant to study a lean implementation of such tools. Our lean implementation has three goals:

- **Easy to program**. Tool developers need not code the parser itself.
- **Easy to understand**. The implemented code should be readable.
- **Practical**. The software tool can be used to test real programs with various syntax components.

The rest of the paper first discusses several potential techniques to implement our testing tools, outlines the chosen approach of utilizing the XML technique, and then uses a case study to illustrate the approach in detail.

II. OUTLINE OF THE LEAN IMPLEMENTATION

A. Main Idea

To achieve the above three goals of lean implementation, we have to utilize existing paring tools rather than developing a parser from scratch, because parser development is still a black art [6].

Reusing and modifying the front-end of an existing compiler is a potential approach. However, the learning curve of the front-ends is steep, and the representations of the abstract syntax tree (AST) generated by different compiler frameworks are quite different. Hence, it is cost-ineffective. Furthermore, when an AST is transformed and unparsed, it is difficult to preserve such features of the original source as space, comments, and preprocessor directives.

On the other hand, we can indirectly reuse the front-ends to achieve our goals by utilizing the XML [2] representation of the source *PUT*. The XML representation is generated by such conversion tools as JavaML [3] and srcML [5]. By modifying the front ends of a compiler, the conversion tool can convert the source code to its XML representation, and vice versa.

Since XML representations can reveal the structure of the source codes very well, we can further use existing extensive XML tools to analyze and manipulate the XML representations according to the type of software testing required.

- If the testing is *static*, we only need to extract the interested information from the XML representation without modifying it.
- If the testing is *dynamic*, such transformations as instrumentation will be made to *PUT*. Hence, we need

This research is supported by the Jinan University Youth Foundation under Grant #51208035, Union Grant of Guangdong Province and National Natural Science Foundation of China under Grant #U0775001, and the Guangdong Province Science Foundation under Grant #7010116.

§ All correspondence should be addressed to Prof. T.H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2858 4141. Email: thtse@cs.hku.hk.

978-0-7695-3336-0/08 $25.00 © 2008 IEEE
DOI 10.1109/CSSE.2008.515

to manipulate and transform the XML representation according to the testing target.

The idea of using XML technologies in developing program analysis tools are also mentioned by the authors of some conversion tools [3, 5, 7]. To the best of our knowledge, however, this paper is the first in discussing its application to both types of software testing, illustrated with a complete case study.

B. Approach

The approach of our lean implementation consists of four steps as follows:

1) Select an appropriate conversion tool according to the language of the *PUT*. The tool should be able to automatically make bidirectional conversions between the source *PUT* and its equivalent XML representation. For example, if the *PUT* is programmed in Java, we can choose JavaML as the conversion tool.

2) Using the conversion tool, *convert source PUT to its equivalent XML representation.*

3) Select an appropriate XML DOM API package according to the programming language of the testing tool. For example, if the testing tool is programmed in Java, we can utilize the APIs supplied by DOM4J [1].

4) By utilizing the API package in *3)*, *manipulate the source XML representation* according to the type of software testing:

a) If the testing is static, extract interested information from the XML representation, according to the testing target. The testing task is complete.

b) If the testing is dynamic,

- Transform the XML representation according to the testing target such as instrumentation, to obtain a transformed XML documentation;
- By using the conversion tool in *1)*, convert the transformed XML document to its equivalent transformed source program;
- Execute the transformed source program, and the testing tasks will be performed.

III. A CASE STUDY

A. Target of GMPS

GMPS [8] is one of the CASE tools of the TACCLE methodology [4] for object-oriented software testing. As a dynamic testing tool, GMPS aims to generate the composite sequences of passing messages and related actions of an object-oriented *PUT*.

GMPS has to deal with the following syntax components of the *PUT*: **conditional statements**, **loop statements**, **return statements**, **method invoking statements**, and **assignment statements of data members of a class**. When the *PUT* is executed, GMPS will monitor all the messages passing among methods and monitor all the modifications of class data members. The intercepted information will be naturally segmented according to the location of information within a condition block or a loop block. Moreover, GMPS also needs to extract block-related components such as conditional predications to mark the blocks.

Our case study develops a GMPS tool for Java source programs. We firstly use the JavaML tool to convert a Java source program (called *Source_Java*) into its XML equivalence (called *Source_XML*), and then utilize DOM4J to parse and manipulate the *Source_XML* to obtain a new XML file (called *Transformed_XML*), followed by the use of the JavaML tool once again to obtain the equivalent Java source program (called *Transformed_Java*) of the *Transformed_XML*. Finally, we execute the *Transformed_Java* and obtain a message-passing sequence of the *PUT*.

B. GMPS and JavaML DTD

Before utilizing JavaML, we have to ensure that JavaML is powerful enough to support the implementation of our GMPS tool. According to the DTD of JavaML, fortunately, we find that all the syntax components described above can be well-depicted by the JavaML representation of the source code under test.

Based on [3], the DTD elements in JavaML corresponding to the above syntax components in Java source are briefly listed as follows:

<!ELEMENT *if* (*test*, *true-case*, *false-case*?)>

<!ELEMENT *loop* (*init**, *test*?, *update**, (%stmt-elems;)?) >

<!ELEMENT *do-loop* ((%stmt-elems;)?, *test*?)>

<!ELEMENT *return* (%expr-elems;)?>

<!ELEMENT *send*(target?, *arguments*)>

<!ELEMENT *field-set*(%expr-elems)>

During the processing of the above syntax components, GMPS also extracts such information as the conditions of conditional or loop statements, and the class names of methods or data members. All the information is also uncovered easily by the JavaML representation. For example, the sub-elements named *test*, *true-case*, and *false-case* of an *if* element in *Source_XML* depict, respectively, the conditional predicate, true branch, and false branch of an **if** statement in *Source_Java*.

C. Key Implementation

The JavaML tool can automatically convert between a Java source program and its corresponding XML representation. Existing DOM4J APIs as well as Java library utilities make the parsing and manipulation of the XML file very easy. Therefore, most of the development of our GMPS tool is programming based on DOM4J APIs and Java libraries, which enables a pretty lean implementation.

In order to use Java utilities and DOM4J APIs, GMPS has to firstly import them. Then GMPS uses DOM4J to parse the *Source_XML* into a Document element as follows:

```
import java.util.* ;
import org.dom4j.*;
String srXML = "d:/source_xml.xml";
SAXReader reader = new SAXReader( );
Document doc = reader.read(srXML);
```

According to the target of GMPS, we can now make transformations to the Document element *doc* that represents *Source_Java*, the Java source code under testing. In GMPS, actually, all the transformations to *Source_Java* are instrumentations. Hence, all the transformations to *doc* should be insertion of elements.

For instance, when processing **if** statements of *Source_Java*, GMPS searches both the true branch and the false branch of each **if** statement, and instruments at the beginning and at the end of each branch. The instrumentation contains the keyword **if** as well as its conditional predicate. The following segment illustrates how to instrument at the beginning of the true branches of **if** statements. When replacing the two annotated statements by its annotations in the following code, the resultant segment can instrument at the end of the true branches of **if** statements. When replacing the "true_case" in the code with a "false_case", the resultant segment will instrument the false branches of **if** statements.

```
List kwList = doc.selectNodes("//if");
for ( Iterator iter = list.iterator( ); iter.hasNext( ); ) {
    Element keyword = (Element) iter.next( );
    String kwName = keyword.getName( );
    // String kwName =  "end_" + keyword.getName( );
    String condition =
        keyword.element("test").getText( );
    Element branch = keyword.elment("true_case");
    List brElements = branch.elements( );
    String instruTxt =
        "<if> <test> bFisrtLoop </test>
        <true_case> <send message = \"write\">
        <target> <var_ref name = \"mp\"/> </target>
        <arguments> <binary_expr op = \"+\">
        <binary_expr  op = \"+\">
        <literal_string value = \" " + kwName + "  \"/>
        <var_ref name = \" " + condition + " \">
        <binary_expr> <literal_string value = \"\n\"/>
        </arguments> </send> </true_case> </if> ";
    Document docTmp =
        DocumentHelper.parseText(instruTxt);
    Element instru =
        (Element) docTmp.getRootElement( ).detach( );
    brElements.add(0, instru);
    // brElements.add(brElements.size( ),instru);
    }
```

In a way similar to the segment above, GMPS instruments **loop statements**, **return statements**, **method invoking statements**, and **assignment statements of class data members**. The primary differences are the searched *keyword*s.

D. Important Details

New GMPS analyzes and manipulates the above syntax components in *Source_XML* one by one. The components analyzed contain condition statements, loop statements, return statements, method invoking statements, and assignment statements of data members. The manipulation is the insertion of statements. If the inserted statements also fall within the analyzed components, measures must be taken to avoid confusions.

In GMPS, all inserted statements fall into two types: conditional statements (**if** statements) and method invoking statements (***mp*.write**). For example, when processing an **if** statement in *Source_Java*, GMPS will insert the following Java code at the beginning of the true branch of the **if** statement:

```
if ( bFirstLoop )
    mp.write ( "if" + condition + "\n" );
```

In order to avoid any confusion between statements being analyzed and the ones being inserted, our GMPS takes two measures:

1) Limit the order of the processing of the syntax components. GMPS deals with conditional **if** statements first, followed by statements of other types.

2) Exclude the processing of special statements. When GMPS analyzes method invoking statements, it will not operate on any method invoking statement starting with "*mp*.write".

E. Analysis of Empirical Results

The original GMPS tool [8] was programmed in C++ using interpreting technique. The development time for the core code (excluding the code for GUI) was about 3 months by one programmer, and the size of the core code was 113 KB. Since it is difficult to develop a full-fledged interpreter for an object-oriented program, the old GMPS was a prototype tool which could only process a limited number of syntax components of the *PUT*.

The new GMPS tool in the present case study utilizes the JavaML tool and is programmed in Java based on DOM4J APIs. The JavaML tool is used to convert *Source_Java* to *Source_XML*, as well as converting *Transformed_XML* to *Transformed_Java*. We do not need any programming during these two conversions, because it is done automatically by the JavaML tool.

The programmer of our new GMPS only needs to program the transformation of *Source_XML* into *Transformed_XML*. The corresponding code size is about 70 KB, which can be developed within only one month by one programmer.

Although the new GMPS takes less time to develop, it can process any Java *PUT* without syntax limitation. This is because the parsing of the program is achieved by the DOM4J

APIs. The liberated GMPS programmer can concentrate on the transformation of *Source_XML*.

The running time of the new GMPS consists of 2 parts: (*t1*) the time for the two conversions using JavaML and (*t2*) the time for the transformation from *Source_XML* to *Transformed_XML*. According to two experiments we performed, *t2* is comparable with the running time of the old GMPS tools. The total time *t1* + *t2* is less than 1 second, still of the same order of magnitude as *t2*.

IV. CONCLUSION

Many software testing tools have to parse the source code before doing the analysis. Parsing programs by compiling or interpreting techniques is classical, but is time-consuming and difficult. This paper illustrates a new approach to simplify the development process.

The case study shows that our three goals in lean implementation of testing tools can be achieved. By utilizing conversion tools such as JavaML, testing tool developers are released from analyzing and parsing the source *PUT*, and only need to program the manipulation of XML documents. By using XML DOM API packages such as DOM4J, programs for manipulating XML documents are readable and easy-to-develop. Since appropriate conversion tools can be used to parse any real-life *PUT*, the testing tools can deal with practical *PUTs* with various syntax components.

REFERENCES

[1] "dom4j 1.6.1 API," available at http://www.dom4j.org/apidocs/.

[2] "Extensible markup language (XML)," W3C, available at http://www.w3.org/XML.

[3] G.J. Badros, "JavaML: a markup language for Java source code," *Computer Networks*, vol. 33, no. 1–6, pp. 159–177, 2000.

[4] H.Y. Chen, T.H. Tse, and T.Y. Chen, "TACCLE: a methodology for object-oriented software testing at the class and cluster levels," ACM Transactions on Software Engineering and Methodology, vol. 10, no. 1, pp. 56–109, 2001.

[5] M.L. Collard, H.H. Kagdi, and J.I. Maletic, "An XML-based lightweight C++ fact extractor," in Proc. of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003), IEEE Computer Society Press, Los Alamitos, CA, pp. 134–143, 2003.

[6] P. Klint, R. Lmmel, and C. Verhoef, "Toward an engineering discipline for grammarware," ACM Transactions on Software Engineering and Methodology, vol. 3, pp.331–380, 2005.

[7] J. Maletic, M. Collard, and H. Kagdi, "Leveraging XML technologies in developing program analysis tools," in Proc. of the 4th International Workshop on Adoption-Centric Software Engineering (ACSE 2004), Edinburgh, UK, pp. 80–85, 2004.

[8] Y.X. Sun and H.Y. Chen, "A new approach and CASE tool for object-oriented dynamic tests at cluster-level with data types of pointer and reference," in Proc. of the 2003 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2003), IEEE Computer Society Press, Los Alamitos, CA, vol. 2, pp. 1075–1080, 2003.

[9] Y.X. Sun and H.Y. Chen, "Use object-oriented paradigm to design and implement an algorithm for object-oriented class-level testing," in Proc. of the 2003 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2003), IEEE Computer Society Press, Los Alamitos, CA, vol. 2, pp. 1069–1074, 2003.