STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LII**, Number 2, 2007

WEB SOURCE CODE POST-PROCESSING: A NEW APPROACH BASED ON CLASSIC MODELS AND METHODS

FLORIAN BOIAN

Abstract. The majority of today's technologies for distributed and web application development adopted programming languages from the C family. To increase the application security, popular such languages like Java, C#, and PHP have been designed without features for direct memory manipulation such as pointers, pointer arithmetic, or memory buffer casting to primitive types similar to the C union construct. The lack of these features makes much easier the implementation of pre-processing and post-processing models for source code simplification, verification and testing. In this paper, a formal approach post-processing mechanism for the web languages is describe. At the moment, the web developing languages has not use the post-processing techniques.

Keywords: source to source transformation, abstract programming schemes, formal methods of source code representation, web applications.

1. Introduction

Source to source code transformation is a wide-spread research direction [1, 11] that studies methods of source code improvement and optimization through automatic manipulation. Source code optimization through refactoring [1, 6] and elimination of redundant control structures are such source to source methods. For instance, sequences like the one in the left side of the table below, can be automatically transformed in the equivalent form on the right, provided that the a and t code segments are independently.

IF t THEN	a;
a;	
b;	IF t THEN
ELSE	b;
a;	ELSE
c;	c;
ENDIF	ENDIF

This optimization method can also be applied to sequences of assignments as shown in the table below. In our example the transformation is conditioned by the independence between e and b*c [15].

`a = b * c + d;`	`temp = b * c;`
`x = b * c / e;`	`a = temp + d;`
	`x = temp / e;`

More recent research directions are focused on automatic parallelization of source code [8]. For example, the code sequence in the left side of the table below can be transformed in the sequence on the right, provided that a(i) and b(i) are independent except for the presence of variable i. In the new form, the two FOR loops in the sequence on the right can be parallelized.

`FOR i ... DO`	`FOR i1 ... DO`
`  a(i)`	`  a(i1)`
`ENDFOR;`	`ENDFOR;`
`FOR i ... DO`	`FOR i2 ... DO`
`  b(i)`	`  b(i2)`
`ENDFOR`	`ENDFOR`

The LOOP-EXIT and LOOP-EXIT-CYCLE [13, 2, 5] schemes revolutionized the automatic source to source transformation techniques. Source code structured using these schemes can be represented easier with formal abstract constructs and thus it is easier to process automatically. These schemes will be addressed in more detail in the following sections.

The transformations presented above are implemented in most of today's compilers.

Today's technologies for development of distributed and web application are based on C-like programming languages without the C features that usually make the code vulnerable. Thus, popular programming languages such as Java [7, 12], C# [14], and PHP [10] have been designed without features for direct memory manipulation such as pointers, pointer arithmetic, or memory buffer casting to primitive types similar to the C union construct. Under these circumstances, many of the problems faced by the source to source transformation methods [1, 11] are no longer possible. Consequently, it becomes much simpler to implement and apply source to source translation for code improvement, optimization, validation, and testing.

In the following sections we will present and discuss source to source transformations using as abstract concept the LOOP-EXIT schemes. The examples will be written in one of the C-like programming languages presented above. The scope of these transformations is to process server side code and detect at an abstract level incoherent code and suggest improvements to the developer. These transformations can be also used to provide information to support server side code execution and logging.

2. LOOP-EXIT Schemes, Branches, and Sections

For the purpose of this paper, we consider the LOOP-EXIT schemes as they are defined in [5]. We denote by A the set of the assignment symbols, and with T the set of the test symbols.

To each LOOP-EXIT scheme S, a language L(S) can be associated. The context-free grammar of scheme S is:

$$G(S) = (N, \Sigma, P, \Delta)$$

Here N is the set of non-terminals, Δ is the axiom of grammar G(S). For each IF_k from S there is a nonterminal I_k in N. For each $LOOP_k$ there are two nonterminals L_k and B_k. Σ is the terminal symbol set. Each symbol from A appears in Σ. For each symbol t from T, the symbols t+ (for true) and t- (for false), appear in Σ. The product rules are detailed in [5,6,8]. In this section we will avoid presenting how the products are formal constructed, but rather will give a practical example in the next section.

In a LOOP-EXIT scheme S, we can define special complete execution paths. A section from S is a maximal sequence in Σ^* where the order of the symbols is the same as in the static text of S.

More exactly, a word z from Σ^* is a section iff exists a word w from L(S) so that :

(1) w = z, or
(2) w = xz and the last symbol from x appears in the text of S after the first symbol from z, or
(3) w = zy and the last symbol from z appears in the text of S after the first symbol from y, or
(4) w = xzy and conditions 2 and 3 are true.

We denote by SECT(S) the set of the sections. A branch in S is a word in SEC(S) so that only conditions 1 or 2 above are true.

We denote by BRAN(S) the set of the branches from S. The sets SECT(S) and BRAN(S) can be generated as regular and finite language with products constructed starting with G(S). The complete construction algorithm is presented in [8].

3. A post-processing example of branch calculation

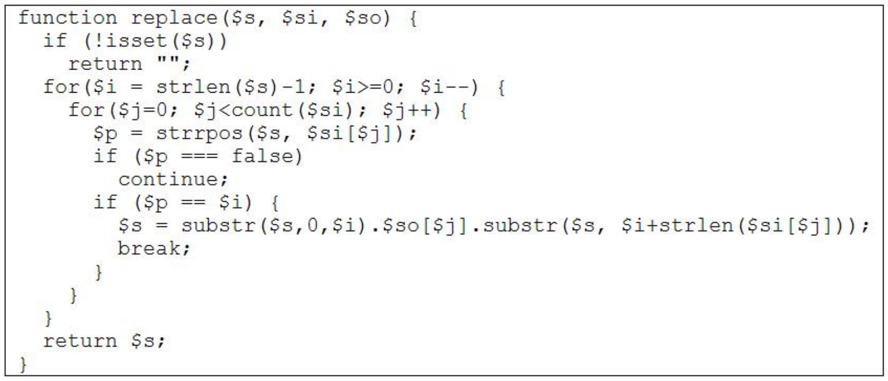
The following PHP function replaces in the input string special HTML characters and non-ASCII characters with their corresponding HTML codes.

The function returns the ASCII string resulted from processing. The arrays of special character and their corresponding HTML codes are:

$sa = array(” ” ,”&” ,”\”” ,”<” ,”>” ,”|” , ...);

$sc = array(”&nbsp;”,”&amp;”,”&quot;”,”&lt;”,”&gt;”,”&brvar;”, ...);

Translating function receives as arguments the input string, and the two arrays above. The PHP code of the function is:



```
function replace($s, $si, $so) {
  if (!isset($s))
    return "";
  for($i = strlen($s)-1; $i>=0; $i--) {
    for($j=0; $j<count($si); $j++) {
      $p = strrpos($s, $si[$j]);
      if ($p === false)
        continue;
      if ($p == $i) {
        $s = substr($s,0,$i).$so[$j].substr($s, $i+strlen($si[$j]));
        break;
      }
    }
  }
  return $s;
}
```

The transformation is executed using the list of variables and constants that appear in the source code.

V = {$s, $si, $so, ””, $i, $j, $p, false}

The assignments statements will be grouped in set A and will be denoted by a1, a2, ..., an. The test statements will form the set T and will be referred to as t1, t2, ..., tn.

The PHP code above will be first translated in abstract code using LOOP-EXIT-CYCLE constructs [13,5]. The result of the transformation is in the following table, the left part.

In the abstract code above, we use EXIT for exiting the inner-most cycle, and CYCLE for jumping to the beginning of the inner-most cycle.

According to theory, a LOOP-EXIT-CYCLE scheme can be automatically transformed in LOOP-EXIT scheme by adding additional LOOP-ENDLOOP statements. The code resulting after these transformations is presented in the following table, the right part. To simplify the code, we replaced the real statements with abstract ones.

```
LOOP1                                  LOOP1
  IF t1($s) THEN                         IF1 t1 THEN1
    a1("")                                 a1
    EXIT                                   EXIT1
  ENDIF                                  ENDIF1
  a2($i, $s)                             a2
  LOOP2                                  LOOP2
    IF t2($i) THEN                         IF2 t2 THEN2
      EXIT                                   EXIT2
    ENDIF                                  ENDIF2
    a3($j)                                 a3
    LOOP3                                  LOOP3
                                             LOOP4
      IF t3($j, $si) THEN                      IF3 t3 THEN3
        EXIT                                     EXIT3
      ENDIF                                    ENDIF3
      a4($p,$s,$si,$j)                         a4
      IF t4($p,false) THEN                     IF4 t4 THEN4
        CYCLE                                    EXIT4
      ENDIF                                    ENDIF4
      IF t5($p,$i) THEN                        IF5 t5 THEN5
        a5($s,$i,$so,$si,$j)                     a5
        EXIT                                     EXIT3
      ENDIF                                    ENDIF5
      a6($j)                                   a6
                                             ENDLOOP4
    ENDLOOP3                               ENDLOOP3
    a7($i)                                 a7
  ENDLOOP2                               ENDLOOP2
  a8($s)                                 a8
  EXIT                                   EXIT1
ENDLOOP1                               ENDLOOP1
```

According to [5], the productions of the grammar associated to scheme S above are:

$\Delta \quad -> $ L1
L1 -> I1 a2 L2 a8 L1 | t1+ | a2 B2 a8
B1 -> I1 a2 L2 a8 B1 | ε
I1 -> t1-
L2 -> I2 a3 L3 a7 L2 | t2+
B2 -> I2 a3 L3 a7 B2 | ε
I2 -> t2-
L3 -> L4 L3 | B4 t3+ | I3a4I4 t5+ a5
B3 -> L4 B3 | ε

L4 -> I3 a4 I4 I5 a6 L4 | I3 a4 t4+

I3 -> t3-

I4 -> t4-

I5 -> t5-

The language generated by this grammar is:

L(S) = t1+a1 | t1-a2(t2+ | t2-a3(t3+ |(a4t4+ | (a4t4-t5+a5 | a4t4-t5-)* a7)*)a8)*

The branch set of scheme S is:

BRAN(S) = {t1+a1, t1-a2t2+a8, t1-a2t2-a3t3+a7, t1-a2t2-a3t3-a4t4+, t1-a2t2-a3t3-a4t4-t5+a5, t1-a2t2-a3t3-a4t4-t5-a6}

The section set of the scheme S is:

SECT(S) = BRAN(S) ∪ {a2t2+a8, a2t2-a3t3+a7, a2t2-a3t3-a4t4+, a2t2-a3t3-a4t4-t5+a5, a2t2-a3t3-a4t4-t5-a6, t3+a7, t3-a4t4+, t3-a4t4-t5+a5, t3-a4t4-t5-a6}

4. Conclusions

The example presented in the section above, shows that post-processing is easily to implement in programming languages lacking direct memory manipulation features such as PHP. Server side applications are usually written in Java, C#, or PHP all of which make post-processing simple. Further optimizations of post-processing can be done on syntactically correct source code. For instance, the post-processing can skip verifying matching parentheses or brackets, or checking for correct statement closing with ";", thus reducing processing work. Post-processing is also simplified by the existence of reserved keywords and variable declaration.

What are the benefits of post-processing? In our view, post-processing in the sense presented above, can assist the developer, in the following ways:

- A tool that can provide the code flow branches can aid the developer visualize the code and place proper logging messages in relevant places.
- According to [9], it is possible to perform an automatic analysis of branches to detect un-initialized variables. Java and C# are able to signal such cases at compile time, while PHP signals such cases only at runtime.
- The branch analysis can help the developer take a series of code refactoring decisions, that remove redundant operations and simplify the flow.
- The representation of the real code into an equivalent abstract code can give the developer a different perspective of the code, which can lead to positive changes in the code.

- Server applications are inherently difficult to debug. Post-processing can help restructure the code in manners making it clearer and easier to analyze, leading to less runtime errors.
- Large scale web applications (such as the one presented in [3]) raise problems more complex than smaller applications. Branch analysis is mandatory to reduce the problems experienced by the large number of users accessing the application, and avoid high maintenance costs.
- Web applications working under high load must be optimize the usage of the resources. The section extraction done by post-processing assist the developer to organize the information to be saved in the HTTP session objects [4].

As a future work, we will implement a post-processing mechanism, for the web languages as PHP, C#, and Java.

References

[1] **Arsac J. J.,** *Syntactic Source to Source Transformation and Program Manipulation.* Comm. ACM, 22, no 1, 1979, pp. 43-53.

[2] **Baker B.S., Kosaraju S.R.**, *A Comparision of Multilevel Break and Next Statements.* Journal ACM, 26, no 3, 1979, pp 555-566.

[3] **Boian F.M. et.al.,** *Distance Learning and Supporting Tools at Babes-Bolyai University* IEEII - Informatics Education inEurope, 29-30 November 2007, Thesaloniki, Greece (accepted - to appear).

[4] **Boian F.M. et.al.,** *Some Formal Approaches for Dynamic Life Session Management* KEPT 2007 Knowledge Engineering Principles and Techniques, Cluj University Press 2007 ISBN978-973-610-556-2, pp. 227-235.

[5] **Boian F.M.,** *Loop - Exit Schemes and Grammars; Properties, Flowchartablies.* Studia UBB, Mathematica, XXXI, 3, 1986, pp. 52-57.

[6] **Boian F.M.,** *Reducing the Loop - Exit Schemes.* Mathematica (Cluj) 28(51), no 1, 1986, pp. 1-7.

[7] **Boian F.M., Boian R.F.,** *Tehnologii fundamentale Java pentru aplicaii Web*, Editura Albastr - grupul Microinformatica, Cluj, 2004.

[8] **Boian F.M., Frentiu M. Kasa Z.,** *Parallel Execution in Loop - Exit Schemes.* UBB, Faculty of Mathematics and Physics, Research Seminaries, Seminar on Computer Science, Preprint no. 9, 1988, pp. 3-16.

[9] **Boian F.M., Frentiu M.**, *Program Testing in Loop - Exit Schemes.* Studia UBB, Mathematica, XXXVII, 3, 1992, pp. 21-30.

[10] **Converse T. et. al.,** *PHP5 and MySQL Bible.* Wiley, 2004.

[11] **Greibach S.** *The Theory of Program Structures: Scheemes, Semantics, Verification.* Springer Verlag, LNCS 1975, 36,1.

[12] **Jendrock E. et.al.,** *The Java^{TM} EE 5 Tutorial, Third Edition: For Sun Java System Application Server Platform Edition* Addison Wesley, 2006.

[13] **Moss C.D.S.,** *Structured Programming With LOOP Statements.* SIGPLAN Not. 15, no 1, 1980, pp. 86-94.

[14] **Turtschi et.al.,** *C# .NET Web Developer's Guide.* Syngress, 2002.

[15] **Vancea A., Boian F.M.,** *On the Exactness of a Data Dependence Analysis Method.* Studia UBB, Mathematica, XLIII, 1, 1998, pp. 13-24.

Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Department of Computer Science, 1 M. Kogălniceanu St., 400084 Cluj-Napoca, Romania

E-mail address: `florin@cs.ubbcluj.ro`