Techniques for Automated Testing of Lola Industrial Robot Language Parser

Maja M. Lutovac and Dragan Bojić

Abstract — **The accuracy of parsing execution directly affects the accuracy of semantic analysis, optimization and object code generation. Therefore, parser testing represents the basis of compiler testing. It should include tests for correct and expected, but also for unexpected and invalid cases. Techniques for testing the parser, as well as algorithms and tools for test sentences generation, are discussed in this paper. The methodology for initial testing of a newly developed compiler is proposed. Generation of negative test sentences by modifying the original language grammar is described. Positive and negative test cases generated by Grow, Purdom's algorithm with and without length control, CDRC-P algorithm and CDRC-P algorithm with length control are applied to the testing of L-IRL robot programming language. For this purpose two different tools for generation of test sentences are used. Based on the presented analysis of possible solutions, the appropriate method can be chosen for testing the parser for smaller grammars with many recursive rules.**

Keywords — **automated testing, grammar-based test generator, parser testing, robot programming.**

I. Introduction

CHECKING the correctness of compiler execution according to the given specifications is a complex task. Compiler should have high reliability and therefore its operations should be carefully verified. Creating an effective set of tests includes an analysis of a large number of combinations. In the context of compiler testing, a test case consists of a test purpose or test case description, a test input consisting of a source program for which the behavior of the compiler is being verified and an expected output which may include a reference to an output file or error file [1]. Also, the testing must cover positive as well as negative test cases [2]. The correctness of parsing affects all other parts of the compiler: verification of semantic constraints, optimizing transformations, and code generation. Therefore, testing the parser provides a basis for compiler testing.

Producing sentences from a grammar, according to various coverage criteria or some other constraints, is required in many applications, such as parser/-compiler testing, natural language processing or grammar validation [3].

In this paper techniques and tools for generating test sentences that are used for automated parser testing are discussed. The analysis of possible solutions is performed and the methodology for testing the parser for smaller grammars with many recursive rules is proposed. Using this methodology, the parser of language for industrial robot programming (Lola Industrial Robot Language - L-IRL) [4], developed at the Lola Institute, is tested.

A technique for parser testing based on production coverage criteria and a technique of generation of positive and negative test sentences are described in Section II. Section III gives a brief overview of algorithms for the generation of test sentences: Grow algorithm, Purdom's algorithm, Purdom's algorithm with length control [5], CDRC-P algorithm [6], CDRC-P algorithm with length control [3] and the algorithm used in the Basil tool [7]. Tools for test sentences generation are described in Section IV. The process of automated testing of L-IRL parser using different algorithms and tools, and obtained results are shown in Section V. Concluding remarks are given in Section VI.

II. Techniques for Parser Testing

A. Grammar-based Testing Technique

Programming language syntax is specified by using context-free grammar which also represents the basis for the construction of the parser within the compiler. A context-free grammar (CFG) is a set of recursive rewriting rules (productions) used to generate patterns of strings.

The grammar-based methodology of testing parsers is based on the use of production coverage as a measure of parser test completeness. By using productions as coverage metric, the process of grammar-based parser testing reduces to identifying a target production, creating a test string that will cause the target production to be applied, and verifying that the target action handling code was run in test. Once a means of running the action handling code has been identified, coverage of the action handler code must still meet target coverage levels [7].

Paper received March 8, 2014; revised May 5, 2014; accepted May 6, 2014. Date of publication July 31, 2014. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Jovan Đorđević.

This paper is a revised and expanded version of the paper presented at the 21th Telecommunications Forum TELFOR 2013.

This work was done under research projects that are supported by the Ministry of Education and Science, Republic of Serbia TR32047 and TR35023.

Maja Lutovac, School of Electrical Engineering, University of Belgrade, Bulevar Kralja Aleksandra 73, 11000 Belgrade, Serbia; Lola Institute, Kneza Višeslava 70a, 11000 Belgrade, Serbia (e mail: maja.lutovac@li.rs).

Dr Dragan Bojić, School of Electrical Engineering, University of Belgrade, Bulevar Kralja Aleksandra 73, 11000 Belgrade, Serbia (e mail: bojic@etf.rs).

B. Generation of Positive and Negative Tests for Parsers

The main idea for the generation of positive and negative tests for parsers is to modify (mutate) the original grammar in order to obtain a language grammar that is similar to the original one but not equivalent to it. These mutant grammars are fed at the input of a test generator, which then produces potentially negative tests [2].

A set of modifications (mutants) is obtained from the component to be tested such that each mutant contains exactly one error. A mutant is said to be killed if it produces a result that differs from the result produced by the initial component. Coverage criterion dictates that all the mutants must be killed. In this case, the mutant grammar will be killed if there is a sequence of tokens belonging to the language specified by the mutant grammar but not belonging to the original language. In terms of parsers, this means that the mutant parser recognizes the sentence and the original parser does not; i.e., the mutant parser turned out to be killed. In the following text the idea of obtaining negative tests for testing parsers by using modified (mutate) sentences of a target language, i.e. by replacing a segment of the sentence with an incorrect one (which also includes an empty segment), is described. First we will define set F_t which is the set of tokens each of which can follow the token t. The complement of the set F_t in the set of terminals T with an empty sequence ε, will be marked as R_t.

$$R_t = (T \cup \{\varepsilon\}) \setminus F_t . \qquad (1)$$

Suppose α is a sentence in the language, and let the token under the number i differ with respect to a given language. A mutation of the first kind, which is denoted by $mut_1(\alpha, i)$, is defined as the replacement of the sentence $\alpha = t_1 \ldots t_n$ by the set of sequences of tokens $t_1 \ldots t_i t' t_{i+1} \ldots t_n$, where t' is an incorrect token from the set R_{ti} ($t' \in R_{ti}$). A mutation of the first kind assumes that incorrect tokens are inserted in a sentence. A mutation of the second kind, which is denoted by $mut_2(\alpha, i)$ is defined as the replacement of the sentence $\alpha = t_1 \ldots t_n$ by the set of sequences of tokens of the form $t_1 \ldots t_i t' t_{i+2} \ldots t_n$, where token $t' \in R_{ti}$. A mutation of the second kind assumes that one of the tokens is replaced by an incorrect token t'.

III. Algorithms for Generating Test Sequences

A. Grow Algorithm

Grow algorithm can be used to generate test sentences for parser testing within the Forson tool. In this algorithm, different rules for each nonterminal symbol possess a certain level of application probability. Associated text is assigned to each symbol and it represents its name. For a "literal", the associated text is simply its name. For a "lexical", the associated text is a random string chosen in the set of lexical values provided in the lexicon input file. If no lexical value is found, its name (as it appears in Bison's token definition) is used as a fallback alternative. Every call of Grow algorithm produces a single syntactically valid sentence of the target grammar. Extra controls must be applied outside the Grow algorithm's implementation in order to reiterate the generation until the requested number of sentences has been generated.

B. Purdom's Algorithm

Purdom's algorithm is a fast algorithm that is useful for parser testing and for debugging grammars [8]. Purdom decomposes the problem of testing parsers in a fashion similar to the grammar-based methodology of testing parsers [7]. The first algorithm is used to find a mapping from nonterminal symbols to strings of terminal symbols, and the second algorithm is used to find a set of derivations that cover all productions in a grammar. Purdom's high level methods are geared toward the verification of parser correctness. The purpose of the algorithm is to generate a set of sentences and the parses of the sentences, such that each production in the grammar is used at least once [8]. The algorithm keeps the total length of the sentences short, but generates them rapidly. The algorithm proceeds in two distinct phases [3]. The first phase statically collects necessary information from the grammar and stores them in some tables. The information includes: length of the shortest sentence that can be derived from each nonterminal and each rule, length of the shortest sentence which uses a nonterminal X in its derivation, which rule to use to derive the shortest sentence from a nonterminal X, which rule to use to introduce a nonterminal X into the shortest derivation, and so on. The second phase dynamically generates sentences by utilizing the information collected in the first phase. A table known as ONCE calculates the next rule to be used for each nonterminal. The algorithm terminates when all the grammar rules have been exploited.

C. The Extension of Purdom's Algorithm

Purdom's algorithm produces too few sentences and some of them are long and complex, i.e., with complicated derivation structures [3]. To avoid the shortcoming of Purdom's algorithm, an improved algorithm is proposed in [5]. It still accomplishes the rule coverage goal but generates more and simpler sentences. The algorithm is built upon Purdom's with two main extensions. First, a reference length is used in the sentence generation process as a reference to control the length of the generated sentence. Second, the length of the shortest sentence derivable from the current derivation is forecasted in the sentence generation process. When choosing a rule to use, it compares this shortest length with the reference length and takes corresponding length control strategies.

D. CDRC-P Algorithm

Modified Purdom's algorithm is proposed in [6]. It is modified by changing table ONCE from calculating the next rule to use for each nonterminal to calculating the next rule to use for each direct occurrence of each nonterminal. In the sentence generation process, it records the context (occurrence) of each encountered nonterminal and then consults table ONCE to choose the right rule to rewrite that nonterminal. When all the rules for all the occurrences in the grammar have been covered, the generation process ceases. The sentences generated by this algorithm are in general more complex than those of Purdom's.

E. The Extension of CDRC-P Algorithm

As a modified version of Purdom's algorithm, CDRC-P preserves most of the features of Purdom's algorithm. It generates relatively few sentences and some of them may be rather long and complicated. Therefore the same length control idea as in the extension of Purdom's algorithm is proposed in [3]. In this way an extension of CDRC-P algorithm generates more and simpler sentences achieving context-dependent rule coverage.

F. The Algorithm used in the Basil Tool

The Basil tool uses the following two-step algorithm [7]:

• Identification of a string in the set of terminals and nonterminals that is guaranteed to exercise some production in production set P.

• Walk of graph Gr (whose vertices represent terminals and nonterminals and transitions are productions in a production set P) starting at the identified string and ending at a string of all terminal symbols.

The resulting string is added to the set of grammar-based tests for grammar G. Generation of a string that exercises some production in P, breadth first search (BFS) algorithm of Gr (G) for a vertex entered by a target production, for each production in P, is used. However, the breadth first search of Gr (G) will commonly cover several productions before a given target production is reached. Redundant computation can be avoided by running a modified breadth first search that searches for a set of targets instead of a single target. Once a set of strings is found, it remains to resolve these strings into strings of all terminal symbols. One solution would be to iterate over each string, performing some search in Gr (G) starting at the string, and ending at the first vertex in Gr (G) that has no out degree. This approach is inefficient because each walk in Gr (G) could follow the same set of productions for common nonterminals in the strings. A solution is to implement a map of a set of nonterminals and to use it to save the obtained path, so that the next time the same nonterminal can be just replaced with a known result.

IV. Tools for Test Sequence Generation

A. Basil

Basil is a tool that enables parser testing by executing grammar-based tests and generation of test strings by using the algorithm described in Section II.A. To enable those functionalities, the test string generator is integrated into the Basil tool [7]. The Basil framework is a Python based set of libraries and applications that are intended to assist software developers in two ways. First, the framework is meant to provide a common infrastructure for programming language implementation. Second, the framework should provide reusable components that allow the development of cross language software engineering and analysis tools.

B. Forson

Forson is a tool for the generation of syntactically correct sentences on the basis of an input Bison grammar file [9]. Any generic Bison grammar file is accepted as an input file. Dynamic data structures are used to avoid restricting the scope of the possible target grammars [10]. Forson contains two operating modes: random and coverage. In the random operating mode, sentences are generated by using the Grow algorithm. In the coverage operating mode sentences are generated by using Purdom's algorithm. In both cases the sentence validity does not exceed the syntactic level. There are no built-in facilities to ensure the semantic conformance to the target language. A partial support to the lexical aspect is instead provided, as it is possible for the user to provide a *.lexicon* input file containing a list of lexical values for the terminal symbols of the target language. In the case of coverage operating mode, a set of sentences is generated such that all grammar rules are used. Minimality of this set often occurs, but is not guaranteed. Forson works under the Linux operating system. It is a batch program. It does not provide (nor needs) an interactive user interface. It serially opens its files, parses its input, elaborates its internal data structure and starts the requested sentence generation. No intermediate temporary files are involved, because the two generation algorithms (Grow and Purdom), ultimately emit the terminal symbols of the target language directly to the output stream.

C. DGL (Data Generation Language)

DGL (data generation language) is a language for automatically generating test data. DGL is based on the concept of probabilistic context-free grammars and allows data to be generated either systematically or randomly, or some combination of the two [11]. The most extensive application of this tool has been in the area of design-verification of VLSI circuits. Although VLSI design verification appears to have very little in common with software testing, such verification is usually done using a software simulation of the circuit rather than the circuit itself. Because of this the differences between software testing and VLSI design verification have all but disappeared.

D. Toolkit for Generating Sentences from Context-Free Grammars

Another toolkit for generating test sentences is proposed in [3]. This toolkit supports sentence generation with coverage criteria, sentence enumeration and sentence analysis. The toolkit deals with general context-free grammars, which have no restrictions on grammars. It consists of several algorithms for sentence generation or enumeration and for coverage analysis for context-free grammars. The sentence generation algorithms provided in the toolkit are based on grammar coverage criteria, including rule coverage (RC) and context-dependent rule coverage (CDRC). For each coverage criterion, two algorithms are implemented, one with a sentence length control mechanism and the other without it. Besides providing implementations of algorithms, it also provides a simple graphical user interface, through which the user can use the toolkit directly.

V. L-IRL (*Lola Industrial Robot Language*) Parser Testing

A. L-IRL (Lola Industrial Robot Language)

L-IRL (*Lola Industrial Robot Language*) is a language for programming and the control of cooperative work of multiple robots, developed at the Lola Institute in Belgrade [4] [12]. L-IRL belongs to the class of problem-oriented languages and contains instructions for programming of robot motions and instructions for programming the logic flow of the program. It enables operations with logical and geometrical expressions as well as operations with environmental signals. It is characterized by structuring and modularity. L-IRL is based on PASCAL programming language with certain modifications. The proposal of the German standard for robotic languages - IRL (DIN 66312) is used as a role model in the realization of modularity and robot-specific program constructions.

B. L-IRL Parser Testing

Testing of L-IRL parser is performed based on test sentences generated by using the Forson tool (using Grow algorithm and Purdom's algorithm) and the tool presented in [3] (using Purdom's algorithm with length control, CDRC-P algorithm and CDRC-P algorithm with length control). It is performed for the correct language grammar, as well as for intentionally modified (mutated) language grammar.

In the case when the Forson tool is used for sentences generation, the file with Bison grammar of the L-IRL language is used as the input file. The list with lexical values connected with the tokens of the target language, in a form of *l-irl.lexicon* file, is used during the testing.

The following text provides an example of a test sentence which is obtained by using Purdom's algorithm, i.e. the coverage operating mode within the Forson tool. Besides the standard instructions for programming the logic flow of the program, the test sentence contains specific L-IRL instructions such as: `wait` instructions (which stop the program execution until a certain condition becomes true), `pause` (which stops the program execution for a specified number of seconds) and `move` instructions that provide programming of industrial robot's motions with additional parameters. `seq` and `endseq` block marks the sequential execution of the instructions.

```
 program p3;
procedure q3 ()
seq
  if 1 then
    case 2 of when 3 :
      for p2 := 1 to 3 step 1
        wait for 2 timeout 2 sec;
            endfor
    default:
      wait for 1 ;
      endcase
  endif;
  pause 3;
endseq
endproc
seq
  return 1;
  move lin 2 act_rob := 3;
endseq
endprogram;
```

An example of test sentence obtained by using random operating mode, i.e. Grow algorithm is given in the following text. The test sentence contains the definition of complex data type that consists of `joint` data type (the data type that describes the position of industrial robot in internal coordinates).

```
program q3 ;
type
  record
  joint [3] : p2, q2, q2, p1, p3, p3, q2;
  endrecord = mainjoint;
endtype;
endprogram;
```

Full coverage is achieved by using Purdom's algorithm. In the case of the application of Grow algorithm to a grammar which has many recursive rules, as is the case with L-IRL grammar, an infinite generation may occur. Forson assigns the same frequency of choice to all the rules of a nonterminal symbol. The only way to try to avoid this behavior is to place extra copies of non-recursive rules in the Bison grammar input file, but this is not an elegant solution [10].

In the case when we used the tool described in [3] for test sentences generation, the input file is the context-free grammar. Once a grammar is input, the editor will check whether it is well-defined. By default, the start symbol corresponds to the left-hand side of the first rule. After generation, the number of generated sentences, the information on sentences lengths, and the total time for generating sentences are also counted. This tool does not contain support to the lexical aspect.

The following text provides an example of a test sentence which is obtained by using Purdom's algorithm with length control. The test sentence contains robot specific data types (`joint` and `addjoint`) that describe robot positions in internal coordinates.

```
PROGRAM IDENTIFIER ' ; '
  VAR
    RECORD
      ADDJOINT LEFT_ARR_BR RIGHT_ARR_BR IDENTIFIER '
      ; '
    ENDRECORD IDENTIFIER ' ; '
    JOINT IDENTIFIER ' ; '
  ENDVAR
  SEQ ENDSEQ
ENDPROGRAM ' ; '
```

An example of test sentence obtained by using CDRC-P algorithm is given in the following text.

```
PROGRAM IDENTIFIER ' ; '
  VAR
    TYPE_REAL IDENTIFIER ASSIGNEMENT NUMBER_INT ' * '
    NUMBER_INT ' ; '
  ENDVAR
ENDPROGRAM ' ; '
```

The following text gives an example of test sentence obtained by using CDRC-P algorithm with length control. L-IRL specific instruction `INV` is used to calculate an inverse value of robot specific data type of `POSE` that describes robot position in external coordinates.

```
PROGRAM IDENTIFIER ' ; '
  SEQ
    PAUSE INV ' ( ' STRING_VAL ' ) ' ' ; ' ENDSEQ
ENDPROGRAM ' ; '
```

Quantitative data on the L-IRL language grammar (number of shifts and tokens) as well as information on the applied mutations are shown in Table 1. Table 1 also shows information about generated positive and negative tests for Purdom's and Grow algorithm, using the Forson tool and positive and negative tests for Purdom's algorithm with length control, CDRC-P algorithm and CDRC-P algorithm with length control (using the tool presented in [3]). The tool presented in [3] can also be used for obtaining the data such as sentences number, total length, average length, maximal length or minimal length.

TABLE 1: QUANTITATIVE DATA ON THE L-IRL GRAMMAR AND PARSER TESTING

Shifts in the grammar	83
Grammar tokens	110
Applied mutations	5
Number of positive tests generated by Purdom algorithm	15
Number of positive tests generated by Grow algorithm	10
Number of negative tests generated by Purdom algorithm	15
Number of negative tests generated by Grow algorithm	10
Number of positive tests generated by RC with length control	69
Number of positive tests generated by CDRC	794
Number of negative tests generated by RC with length control	71
Number of negative tests generated by CDRC	1042
Maximal length using RC with length control for original L-IRL grammar	94
Maximal length using RC with length control for mutated L-IRL grammar	86
Maximal length using CDRC for original L-IRL grammar	3113
Maximal length using CDRC with length control for original L-IRL grammar	106
Maximal length using CDRC for mutated L-IRL grammar	3116
Maximal length using CDRC with length control for mutated L-IRL grammar	109
Semantic coverage in Forson tool	No (partial support to the lexical aspect is provided)
Semantic coverage in tool presented in [3]	No

C. Negative Test Sentences

Generation of negative test sentences is performed by using Grow, Purdom's, Purdom's with length control and CDRC-P and CDRC-P with length control algorithm. The grammar is intentionally "corrupt" (mutated) in order to generate syntactically incorrect sentences. Negative test sentences are obtained using simple mutations, since according to [13]: if simple mutations are detected, then the complex mutations will be also detected.

L-IRL grammar is mutated using the next five mutations: in the rule for defining constants the symbol ':' is inserted behind the terminal symbol CONST; in the rule for defining functions the terminal symbol IDENTIFIER is duplicated; in the rule for defining the var part of the program the symbol ';' is inserted behind the terminal symbol VAR; in the rule for defining new types the symbol '=' is replaced with symbol '<>'. In the rule for stop statement the terminal symbol PAUSE is replaced with terminal symbol PAUSE2. The first, second and third mutations are mutations of the second kind. The fourth and the fifth mutations represent mutations of the second kind. Those mutated grammar rules are given in the next text.

```
  CONST ':' constant_definition_list ENDCONST ';'
  variable_definition_part     :     VAR     ";"
definition_list ENDVAR ';'
  function_definition   :   FUNCTION   IDENTIFIER
IDENTIFIER function_block ENDFCT ';'
  type_definition : type '<''>' type_identifier
  PAUSE2 [ expression ] ';'
```

A potentially negative test sentence that includes the third mutated grammar rule, obtained by using Forson tool and Purdom's algorithm, is given in the following text. The test sentence contains `move` and `move_inc` instructions that describe industrial robot movement (circle, ptp – point to point or lin - linear) with additional parameters. Those two instructions specify the coordinate system in which the motion is programmed. It can be the user coordinate system (`move` instruction) and the tool coordinate system (`move_inc` instruction).

```
  program p1 ;
    function q2  p2 ( ) : real
      seq
        move circle 1, 2 c_cp;
        move_inc  ptp   3 c_ptp;
      endseq
    endfct
    seq
      move_inc lin 3 c_cp := 2;
    endseq
  endprogram  ;
```

A potentially negative test sentence obtained by using the Forson tool, Grow algorithm and the first mutated grammar rule, is given in the following text.

```
  program p3 ;
    const : event : q3 := true; endconst
  endprogram;
```

The following text provides two examples of negative test sentences obtained by using the tool proposed in [3] and Purdom's algorithm with length control. The first example includes the second mutated grammar rule. The second example includes the fourth mutated grammar rule.

```
  PROGRAM IDENTIFIER ' ; '
    VAR ' ; '
      STRING IDENTIFIER ASSIGNEMENT NUMBER_REAL ' , '
      IDENTIFIER ASSIGNEMENT PI ' , ' IDENTIFIER ' ; '
    ENDVAR
    PROCEDURE IDENTIFIER ' ( ' ' ) '
    ENDPROC
    SEQ ENDSEQ
  ENDPROGRAM
  PROGRAM IDENTIFIER ' ; '
    TYPE
      RECORD TYPE_INT IDENTIFIER ' , ' IDENTIFIER ' ; '
      ENDRECORD ' < ' ' > ' TYPE_BOOL ' ; '
    ENDTYPE ' ; '
    VAR ' ; '
      OUTPUT IDENTIFIER AT NUMBER_INT
    ENDVAR
  ENDPROGRAM
```

Examples of negative test sentences obtained by using CDRC-P algorithm with length control are given in the following text. The first example includes the second mutated grammar rule and the second example includes the fifth mutated grammar rule.

```
PROGRAM IDENTIFIER ' ; '
  VAR ' ; '
    BOUNDS   LEFT_ARR_BR   STRING_VAL   RIGHT_ARR_BR
    IDENTIFIER ASSIGNEMENT NUMBER_INT ' ; '
ENDVAR
  FUNCTION IDENTIFIER IDENTIFIER ' ( ' ' ) ' ' : '
IDENTIFIER
    PAR ENDPAR
  ENDFCT
ENDPROGRAM

 PROGRAM IDENTIFIER ' ; '
  SEQ PAUSE2 IDENTIFIER ' ; ' ENDSEQ
ENDPROGRAM
```

D. *Checking the Obtained Results*

Test sentences (positive and negative) which are obtained by using Purdom's algorithm, Grow algorithm, Purdom's algorithm with length control, CDRC-P algorithm and CDRC-P algorithm with length control are compiled by using L-IRL IDE (robot programming environment) and the obtained results are compared to the expected results. The result of compilation for all positive test sentences, obtained by using the Forson tool and the tool proposed in [3] was successful parsing without error notifications. For potentially negative test examples, obtained by using the Forson tool, the compiler has reported the expected syntax errors. For the sentences which are obtained by mutating grammar rules the compiler reported an error: `Syntax error at line X,` where X was the number of the line in which the error was expected. Compiling of negative test examples was also performed for the sentences obtained by using the tool proposed in [3]. Previously, the token values had to be replaced with the possible lexical values. As the result of compiling those test examples, an expected syntax error was reported.

VI. Conclusion

This paper presents a systematic approach to language parser testing, based on available techniques. Using these techniques and tools the parser of language for industrial robot programming is tested. The testing is performed based on positive and negative test examples. For the testing, several algorithms are used: Purdom's and Grow algorithm as a part of Forson tool and Purdom's algorithm with length control, CDRC-P algorithm and CDRC-P algorithm with length control. Grow algorithm was not adequate for testing the L-IRL parser because it doesn't have facilities for limiting the size of a randomly generated sentence. When using this algorithm for the testing of grammar that has many recursive rules an infinite generation may occur. Testing the L-IRL parser using Purdom's algorithm enables complete coverage of all grammar rules. This algorithm may be applied for testing parsers of both large and small programming languages, but it is only complete in the case of a parser for smaller grammars such is L-IRL grammar. The Forson tool is applied for testing with those two algorithms. For the testing by using Purdom's algorithm with length control, CDRC-P algorithm and CRDC-P algorithm with length control the tool proposed in [3] is used. Unlike the Forson tool that is a batch program this tool provides a graphical user interface. Besides the functionality of sentence generation, it provides sentence enumeration and analysis, and quantitative data for used grammars, such as sentences number, total length, average length, maximal length or minimal length. An extension of Purdom's algorithm implemented within this tool accomplishes the rule coverage goal but generates more and simpler sentences than the Purdom's algorithm. CDRC-P algorithm with length control produces more and simpler sentences, achieving context-dependent rule coverage, than the extension of the Purdom's algorithm. Both used tools do not provide semantic coverage of the target language. The advantage of the Forson tool is that it has partial support to the lexical aspect, but in both cases there is a need for manual adjustment of tests sentences. Therefore we propose using shorter test cases.

The paper provides an overview of the techniques and methods used in practice, and, after an analysis of possible solutions, proposes a methodology for initial testing of a newly developed compiler. Based on the presented analysis, the appropriate method can be chosen for testing the parser for smaller grammars with many recursive rules.

References

[1] A. S. Boujarwah, K. Saleh "Compiler Test Case Generation Methods: a Survey and Assessment," *Information and Software Technology*, Vol. 39, 1997, pp. 617–625.

[2] S. V. Zelenov, S. A. Zelenova, „Generation of Positive and Negative Tests for Parsers", *Programming and Computer Software,* 2005, Vol. 31, No. 6, pp. 310-320.

[3] Z. Xu; L. Zheng; H. Chen, „A Toolkit for Generating Sentences from Context-Free Grammars," *8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, 13-18 Sept. 2010, pp.118-122.

[4] M. Pavlović, "High Level Programming Language for mult-robotic operations," (in Serbian), M. Sc thesis, University of Belgrade, Scholl of Electrical Engineering, 1994.

[5] L. Zheng, D. Wu, "A sentence generation algorithm for testing grammars," *Proc. of the 33rd Annual International Computer Software and Applications Conference,* Vol. 1, 2009, pp. 130-135.

[6] Y. Shen, H. Chen, "Sentence generation based on context-dependent rule coverage," *Computer Engineering and Applications*, Vol. 41, No. 17, 2005, pp. 96-100.

[7] J. Riehl, *„Grammar Based Unit Testing for Parsers,"* M.Sc.Thesis, Department of Computer Science, University of Chicago, 2004.

[8] P. Purdom, „A sentence generator for testing parser," *BIT Numerical Mathematics,* Vol. 12, 1972, pp. 366-375.

[9] *Bison - GNU parser generator*, Available: www.gnu.org/software/bison/, June 2013.

[10] *Forson, a sentence generation tool,* Available: http://forson.sourceforge.net/, June 2013.

[11] P. Maurer, „The Design and Implementation of a Grammar-based Data Generator," *Software Practice and Experience*, Vol. 22, pp. 223–244, Marc. 1992.

[12] M. Lutovac, G. Ferenc, V. Kvrgić, J. Vidaković, Z. Dimić, „Robot programming system based on L-IRL programming language," *Acta Technica Corviniensis – Bulletin of Engineering*, Fascicule 2. April–June, pp. 27-30, 2012.

[13] A. J. Offut, R. H. Untch, "Mutation 2000: Uniting the Orthogonal," *Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, Calif., 2000, pp. 45–55.