SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 15, No. 1, February 2018, 71-83

UDC: 004.415:004.43XML **DOI: https://doi.org/10.2298/SJEE1801071P**

Prototype Implementation of Segment Assembling Software

Đorđe Pešić[1,2], Marko Mišić[2], Jelica Protić[2], Milena Vujošević Janičić[3]

Abstract: IT education is very important and a lot of effort is put into the development of tools for helping students to acquire programming knowledge and for helping teachers in automating the examination process. This paper describes a prototype of the program segment assembling software used in the context of making tests in the field of algorithmic complexity. The proposed new program segment assembling model uses rules and templates. A template is a simple program segment. A rule defines combining method and data dependencies if they exist. One example of program segment assembling by the proposed system is given. Graphical user interface is also described.

Keywords: Time complexity, Automated source code assembling, XML technologies.

1 Introduction

There is a big number of students at introductory programming courses at the School of Electrical Engineering, University of Belgrade. The current trend is that the number of students will grow even more in the following years. Automated test assembling and automated evaluation of such tests are of crucial importance for handling both a big number of students and a big number of exams during the year.

There are different solutions for automated test assembling and evaluation. One solution for automated test assembling is to choose questions from the database, taking care about various parameters, like examination area, question difficulty, appearance on the previous exams, etc. In this case, the software can be commercial or freely available and often uses artificial intelligence, logic programming and genetic algorithms [1]. Another solution is question parametrization [2]. The most simple is the numeric parametrization. In programming questions, it can be done with the change of the input data, which leads to the change of the output. Question parts which will be parameterized

[1]RT-RK Institute for computer-based systems, Novi Sad, Serbia; E-mail: djordje.pesic@rt-rk.com
[2]University of Belgrade, School of Electrical Engineering, Bulevar kralja Aleksandra 73, 11120 Belgrade, Serbia; E-mails: marko.misic@etf.bg.ac.rs, jelica.protic@etf.bg.ac.rs
[3]University of Belgrade, Faculty of Mathematics, Studentski trg 16, 11000 Belgrade, Serbia; E-mail: milena@matf.bg.ac.rs

must be marked in some way and the concrete parameters can be chosen randomly by the software.

The evaluation process may be automated either the examination is done on paper or online. In the case of examination on paper, answers are written on special paper forms, which are then scanned and further processed by specialized software [3]. The examination can also be done on the computer (online testing), and there are different techniques for automated evaluation [4].

Understanding basic concepts about algorithm complexity is an important part of introductory programming courses. This paper presents a software tool for source code segments assembling. The time complexity of the assembled segments is further analyzed. The segments can be used for making the exam questions and potentially for learning and understanding the time complexity. The basic motivation for the development of this tool is described in [5], while the XML [6] model of the program segments is presented in [7]. The tool is still under development, so this paper will present developed parts and will discuss the further work.

The paper is organized as follows. Chapter 2 introduce some theoretical basics about time complexity. Chapter 3 describes the connection between the Halting problem and the time complexity. Chapter 4 introduces a conceptual solution. Chapter 5 describes the implementation. Chapter 6 has an example of the software work while Chapter 7 concludes the paper.

2 Time Complexity

Algorithm complexity analysis is an important part of theoretical computer science which studies algorithm time and space complexity [8]. Time complexity models the time needed for the execution of the function of the dimension n, where n represents the problem size. The problem size is usually input data size. Space complexity models the size of memory needed for the execution. Space complexity analysis is similar to time complexity analysis, but it will not be covered in this paper.

Complexity function $T(n)$, which is an approximation of the execution time, is defined as a number of elementary instructions, which have constant execution time. Unfortunately, for the slightly more complex source code segments, this function becomes very complex. Therefore, exact complexity function is usually replaced by the asymptotic complexity function, which behaves the same for sufficiently large *n*. Then, only dominant part of the complexity function remains. For example, for $T(n) = n^2 + 3n + 4$, the quadratic part is dominant so the asymptotic complexity order is $\mathrm{O}(n^2)$. The big O notation formalizes such results and it denotes the complexity upper limit. For any function $T(n)$, $T(n) = \mathrm{O}(f(n))$ if $\exists c, n_0$, $\forall n > n_0$, $T(n) < cf(n)$, then $\lim\limits_{n\to\infty} \frac{T(n)}{f(n)} = c$, $c > 0$.

When this notation is used, the analysis of an algorithm is reduced to the analysis of the general structure of an algorithm, which is far easier than exact analysis. There are some rules for the big O notation. Let two source code segments have complexities $\mathrm{O}(f(n))$ and $\mathrm{O}(g(n))$. The complexity of the sequence is $\mathrm{O}(\max(f(n), g(n))$ [9]. $\mathrm{O}(f(n))$ is greater than $\mathrm{O}(g(n))$ if the following formula is correct:

$$\lim_{n\to\infty} \frac{\mathrm{O}(f(n))}{\mathrm{O}(g(n))} = \infty . \tag{1}$$

When the segments are nested, complexity is $\mathrm{O}(f(n) * g(n))$ [9]. These formulas are valid when the two segments are mutually independent [8].

In this paper, it is assumed that every generated source code segment will have the complexity which depends on one input parameter. This assumption is made in order to make the complexity calculation and the software development easier, but it does not affect the generality of the approach [8]. During the complexity calculation, the program is represented as a set of instructions, where each instruction has one abstract time unit of execution [8]. The instruction can be arithmetic or logic operation, value assignment, branching, etc.

2.1 Halting problem and the complexity calculation

The main idea of our approach is to assemble source code segments for which the time complexity calculation is possible. In the general case, time complexity calculation of the arbitrary source code is an undecidable problem. This is a direct consequence of the halting problem [10]. In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, without any resource or time limits, whether the program will finish running or continue to run forever. Alan M. Turing was the first who proved that this problem is undecidable [10]. Since it is not possible to construct an algorithm that will determine if an arbitrary program will stop its execution, then it is also not possible to construct an algorithm that would count the number of executed statements, i.e. which would calculate the time complexity of an arbitrary program. Therefore, we must introduce some constraints in order to be able to calculate time complexity. In our software, these constraints are connected to the structure of the source code and the complexity of expressions inside.

3 Overview of the Design Concepts

Source code segment is created as a combination of the building blocks. This approach is introduced in this paper, and it provides a greater flexibility, compared to previously used approaches [[5], [7]. Previous approach implements combining as an assembling strategy. The assembling strategy is the set of parameters which can impact the final segment complexity: Chosen combining method, chosen loops types and interaction between them [5]. This approach is not flexible. One strategy definition contains the complete description of combining methods and connections between ports. Also, one definition is stored in a single file. Any change results in the new strategy, even if the final complexity remains unchanged. For example: if the two strategies differ in the expressions which do not affect the complexity, then these strategies should be the same, while the old system sees them as two different strategies. Also, if the two strategies have only variable names different, again, these should be the same strategy. The strategies are now defined in the more generic and more flexible way.

The basic building blocks are empty loops, expressions, and selection statements. Assembled source segments can be further combined among themselves. There are two types of combining: sequencing and nesting. These types of combining can be done in two ways:

1. Combining two mutually independent segments. Two segments are mutually independent if there is no data dependency between them. In this case, if segments are sequenced, time complexity is calculated as the maximum of the two initial complexities. If the segments are nested, the resulting complexity is calculated as the multiplication of the initial complexities. Theory for this calculations is presented in Chapter 2.
2. Combining two mutually dependent segments. These dependencies are created by the user and they are the result of the user's intellectual work. The dependencies are actually defined by the connections between segment ports [7]. Source code segment’s port is a variable, whose value is changed inside the segment’s body, a variable which is placed inside the loop bound expression or a variable that serves as a loop iterator. Time complexity calculation for this case is very difficult and we must introduce various constraints, in order to make the calculation possible. This is the subject of our future work.

The system should be able to generate various source code segments, using the set of available segments and combining operations. As an error prevention, time complexity should be calculated by the software, if possible.
If we want to make the automatic complexity calculation possible, source code segments must be abstracted in some way. A.W. Bierman presented one

abstraction of the source code segments [11]. He presented the segments as a sequence of the abstract expressions, which are created from the set of the rules, applied to the concrete source code. He developed this approach to help his students in better understanding of the time complexity calculations. Some of the ideas are used in this paper.

This paper uses the reverse approach: starting from the set of defined templates and rules, templates are combined with the rules and, as a result, the segments are assembled. As a benefit, there is a trace of assembling steps. The trace contains used operations and defined data dependencies. If the time complexity of the templates is known, in some cases, the complexity of the assembled segments can be determined by the software.

In the current software version, time complexity can be calculated when the segments which are combined does not have any data dependency. Then, in the case of nesting, result complexity is the product of the combined complexities. In the case of sequencing or selection, result complexity is calculated according to (1).

The result of the new software concept is a greater set of source segments which can be assembled, and the possibility to present the assembling process, which can be used in learning. Students can analyze the presented process and better understand the time complexity. Another benefit is the template and rule reusability. Also, the software can expand its template database, because every assembled segment has template format and can be added to the template database.

3.1 Templates and rules

Templates are basic building blocks which are used in assembling process. Basic templates are an empty for loop, an empty while loop, an empty repeat loop, an empty if statement and an expression. These basic templates are recognized from the imperative programming. Control flow methods are sequencing, selection, iteration, procedural abstraction, recursion, concurrency and exception handling [12]. Sequencing, selection, and iteration the most important for the time complexity analysis in our context. Iteration is achieved with the loop templates. In general, templates can be arbitrary complex, but it is strongly recommended to keep the templates simple, to maintain the reusability.

The rules describe which operations are used for the template combining, which templates are used and define the data dependencies between them. Also, rules can combine the outputs from the other rules. One rule can work with two templates, one template, and one rule output or two rule outputs. This enables rule cascading during the segment assembling. Rules are also stored in the software's database. When the concepts of the rules and templates are defined, it is possible to assemble source code segments in a controlled manner, where the time complexity of the resulting complexity can be calculated.

3.2 Graphical user interface

Graphical user interface is a new part of the software, which does not exist in the previous versions, described in [5] and [7]. The main goal of the graphical interface is to make the interaction with the user easier and to make the testing easier. Also, the graphical interface has a part for template and rule management. In the future work, the graphical interface will have a part for managing the assembling process (guided or totally automated assembling) [13]. Guided assembling is the process when a user provides the target complexity and then, the software guides the user through the assembling. Automated assembling is the process when the user provides the target complexity and then, the software chooses which templates and rules should be used in order to achieve the wanted complexity.

4 Implementation

This chapter will describe the software implementation. The implementation will be presented from the lowest layer (source segment representation, to the highest layer (graphical user interface).

As source segments should have an arbitrary complexity and structure, a subset of the EBNF Pascal grammar [14] is used, and class hierarchy, which represents this grammar subset, is implemented. The result of the assembling process is a syntax tree. The tree elements are instances of the classes which represent grammar terminals and nonterminals. The concrete source code is generated with the recursive call of toString() methods. The last step in source segment modeling is the segment port definition. Ports can be input (the variables that influence the observed loop) and output (the variables that from the observed loop influence some other loop). Source code segments are stored in the XML files.

As XML files must be saved to disc and later loaded, the JAXB [15] is used. Because the JAXB is integrated with the Netbeans, there is no need to write the classes, representing the object-to-XML mapping. These classes are generated by the Netbeans from XSD files. XML file unmarshalling procedure also validates the input XML file. If the input file is invalid, the unmarshaller will throw an exception and stop the unmarshalling. When the unmarshal is successfully finished, a tree of JAXB-generated class instances is returned as a result. This tree is an object representation of the XML file.

The syntax tree is generated from the JAXB object tree. It is traversed and the syntax tree is constructed from the data from the JAXB objects. JAXB object trees structure defines the syntax tree structure. The JAXB class hierarchy is recursive, so the functions for JAXB object processing are also recursive.

4.1 XML model of templates and rules

The template is a source code segment. Every template is stored in a separate file, and the files with the templates are stored in a separate directory. Templates can be parameterized, in order to enable the source segment obfuscation. Loop bounds and iterators can be omitted, and then, they are chosen randomly. Variables inside the expressions can be replaced by the special character, and during conversion from XML model to the source code, these special characters are replaced with random variable names. Because the expressions can be arbitrary complex, an ANTLR [16] expression grammar is written and the expression parser is generated. The grammar is expanded with the special character for expression parametrization. When template source code is defined, all variables that appear in it are considered formal variables. When the template is instantiated, the formal variables can be replaced with some concrete actual values.

The rules are also stored in the XML files. Every rule has its own file, and the rule files are stored in a separate directory. Rule format is defined with the XML Schema language. Rule definition has three major parts: Two template definitions, operation definition, and data dependencies definition.
Template definition can be realized in two ways:

1. The file name, template time complexity and formal to actual variable mappings.
2. With the rule name. When this option is used, another rule output is used as a template. This enables rule chaining.

Operation definition contains operation type (sequencing, nesting, and selection) and connections. Connections implement data dependency definition with mapping the input ports to output ports.

4.2 Rule processing

The rule processing is the core of the system. It produces the result source code segments. The result is in the form of the tree of the JAXB-generated class objects. This tree can be forwarded to some other rule or can be used for the final source generation. Rule outputs can be saved in the XML files. The rule processing algorithm:

```
ruleProcessing(ruleXmlModel){
  if(firstTemplateName is ruleName){
    rule = loadRule(firstTemplateName)
    firstTemplate = ruleProcessing(rule)
  } else
    firstTemplate = loadTemplate(firstTemplateName)

  if(secondTemplateName is ruleName){
    rule = loadRule(secondTemplateName)
    secondTemplate = ruleProcessing(rule)
  } else
    secondTemplate =
    loadTemplate(secondTemplateName)

  switch(operation){
    case sequence:
      result = sequence(firstTemplate,
      secondTemplate)
    case nesting:
      result =
      nesting(firstTemplate, secondTemplate)
    case selection:
                  selectionTemplate =
                  loadTemplate(selectionTemplateName)
      result = selection(firstTemplate,
      secondTemplate, selectionTemplate)
  }
  calculateComplexity()
  connectPorts()
  return result
}
```

The first step is to load the rule from the XML file. After that, algorithm checks if the first operand is the template or the rule. In the first case, the template is loaded from its XML file, in the second case, the specified rule is loaded, executed and the rule’s output is forwarded to the current rule. The same procedure states for the second template. During the template loading, formal variables used in the template are replaced with the actual variables, specified in the rule. Thanks to this feature, the same template can be instantiated in the different rules. After the first and the second template loading, the specified operation is executed. The operation is realized with connecting the two object trees, in such manner, that the result object tree also satisfies the XML object tree structure. After the result object tree creation, result complexity is calculated. In the nesting case, result complexity is multiplicand of the two complexities. In the sequencing and the selection case, result complexity is the maximum of the two complexities and it is calculated according to (1). Both calculations are done symbolically, using the Yacas [17] algebra engine, integrated with the system as a Java library.

The last step in the rule execution is the data dependencies realization if they exist. These dependencies are created with the port connections. The rule specifies, which port from the first template should be connected to which port from the second template. Connections are realized by variable renaming. Variables that represent input ports are renamed with the variables representing the output ports. Let the first template have an output port “a”, and the second template have the input port “b”. The rule specifies that “a” should be connected to “b”. Each appearance of “b” in the second template should be replaced with “a”. Such renaming requires traversal of the whole object tree and their changing under some circumstances. This procedure is made easier by XPath

[18]. The XPath is a query language for searching the XML files. In our implementation, the JXPath library is used, because it implements XPath queries on the arbitrary object trees (not only XML files), and also makes the change of the searched trees possible. The XPath itself provides read-only traversals. The rest of the software uses XPath queries whenever the need for traversing and changing the object trees appears.

4.3 Graphical user interface

The graphical user interface (GUI) is the highest software layer. The graphical user interface uses JavaFX [19] technology. The main GUI window contains two tabs. The first tab contains the interface for template management (Fig. 1), the second tab contains the interface for rule management (Fig. 2). The interface for template management contains command bar with the commands and three big panes. The left pane serves as a pane for interactive template creation, the generated XML model of the template will be displayed in the upper right pane and the appropriate source code will be displayed in the bottom right pane.

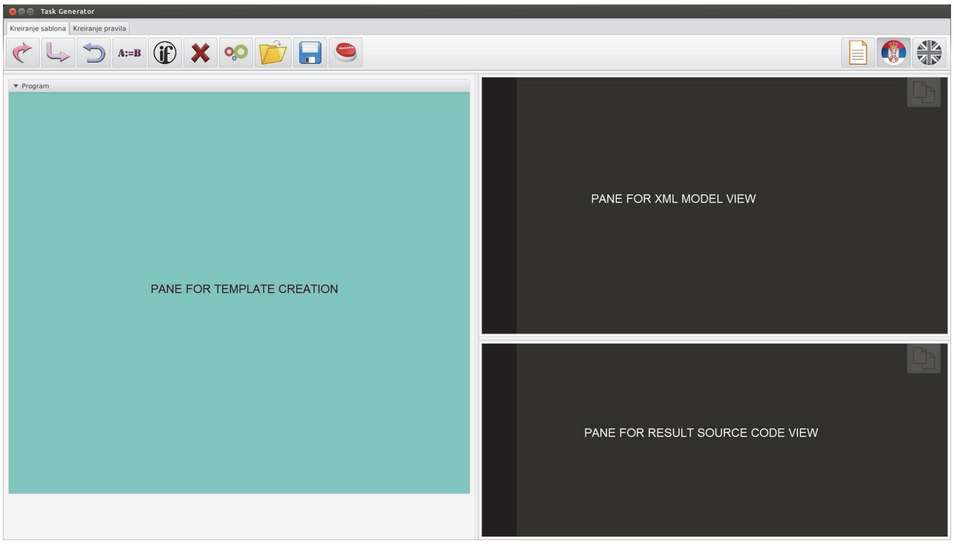


Fig. 1 – *Window for template management.*

The two right panes have the buttons for copying their textual contents. Command bar is divided into two areas. The left area contains commands for template management and the right area contains localization commands and the command for saving the generated source code. Localization currently supports English and Serbian languages, and it can be easily extended to any other language. The command for the template management (from the most left to the right) are: Add for loop, add while loop, add repeat loop, add expression, add empty if statement, delete the segment, generate, load template from the file, save the template to file and delete template file. The command for template addition has three steps: the first is to press the command button (enable), the second is to click inside the left green pane and the third is to click the command button again (or click some other command button) to disable the current command. When the user clicks inside the green pane, a graphical representation of the template is generated. Graphical template representations can be nested or sequenced. When the user clicks inside the left pane, graphical representation appears on the clicked location. If the clicked location is inside the other template, then the new template is nested inside the clicked. Else, the sequence of the graphic elements is created. Graphical representations of "if" statements are created with nesting inside the "if" or "else" branches, or leaving the graphical "if" template empty. Some of the parameters can be omitted and these are later chosen randomly. Template assembling is finished with the click of the "generate" button. Fig. 3 shows an example of the complex template.

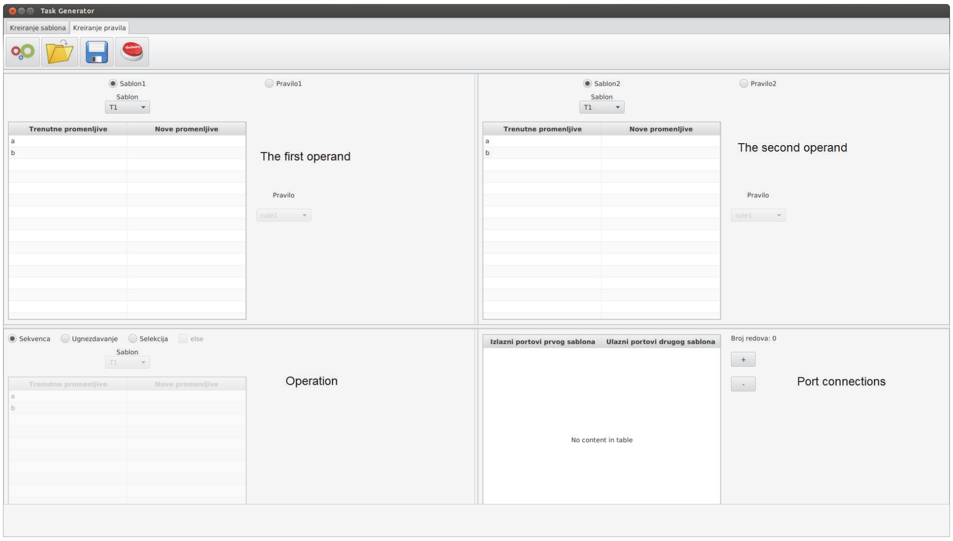
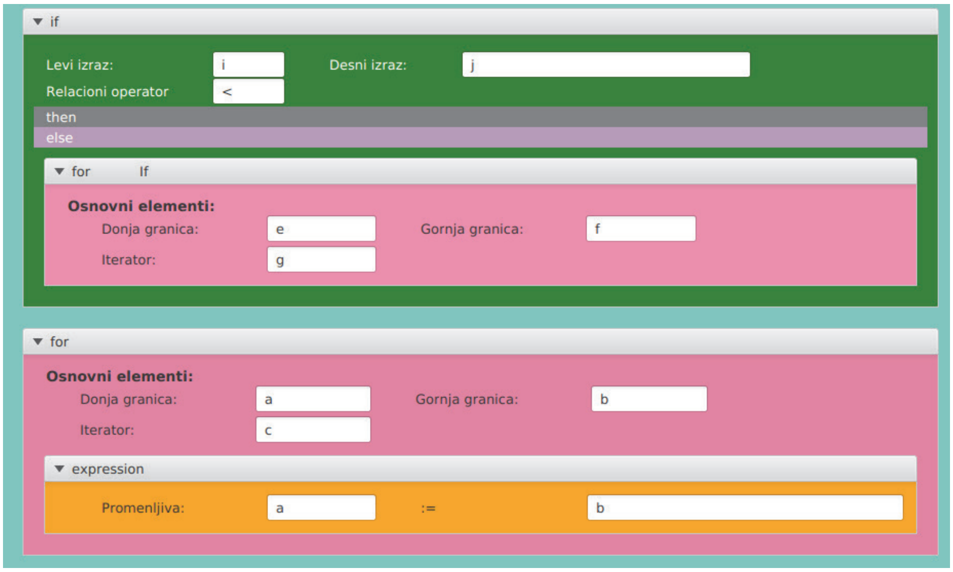


Fig. 2 – *Window for rule management.*

The result is an XML model, shown in the upper right pane. For the sake of completeness, a source code is generated from the XML model and presented inside the bottom right pane. XML model pane is placed inside the scroll pane, and tag collapsing is supported. This provides the easier view of the XML model.

The interface for rule management has the command bar and four panes. Command bar has commands for rule generation, rule loading, rule saving and rule deleting. The upper left pane contains the GUI for the first operand definition. The user selects the radio button if the operand is a rule or a template. If the template option is selected, then the template name is chosen from the combo box and the table with formal to the actual variable map is filled. If the rule option is selected, then the rule name is chosen from the combo box. The same procedure stands for the second operand definition (upper right pane). The bottom left pane contains GUI for operation definition. The user can select sequence, nesting or selection. If the selection is selected, the user can check if "else" branch is present, and chooses the template with the "if" statement. At the end, data dependencies are defined in the bottom right pane. The user enters the names of the ports which should be connected.



▼ if

Levi izraz: i   Desni izraz: j

Relacioni operator <

then

else

▼ for   If

Osnovni elementi:

Donja granica: e   Gornja granica: f

Iterator: g

▼ for

Osnovni elementi:

Donja granica: a   Gornja granica: b

Iterator: c

▼ expression

Promenljiva: a := b

Fig. 3 – *A complex template.*

5 An Example

This chapter will present an assembling example. The example will demonstrate guided synthesis. The target complexity is O(n) and assembling process will use 2 templates and 1 rule. The set of the templates is given in **Table 1**, while the rule is: R1(nest, T1, T2). It nests template T2 inside the T1.

Table 1
Used templates.

Template	Source
T1(i,a,b)	for i:=a to b do;
T2(a,b,c)	a:=b+c

The specified templates are connecting with the specified rule. Execution starts at the tree root and continues recursively. Every template is instantiated with the actual variables. The rule defines formal to actual variable mappings. The actual value appears during the instantiation, while the formal value is defined during template definition. Assembling starts with the rule execution. First, T1 is instantiated and becomes T1(i, 1, N). Then, T2 is instantiated and becomes T2(s, s, 2). After that, nesting is done and T2 appears inside the T1.

The final result is:

```
for i:= 1 to N do
begin
  s := s + 2
end;
```

6 Conclusion

This paper presents the prototype of the software for source code segment assembling. The software contains multiple layers: the layer for converting the XML model to the actual source, the XML data model layer, the source segment abstraction layer, the rule and template layer and the graphical interface. The rule and template layer contains the time complexity calculation when the data dependencies do not exist. The output is the source code segments, which may be used as exam questions.

The future work will be oriented to the time complexity calculation improvements when the data dependencies are present. There is a need for the thorough research because this problem is theoretically undecidable. The second research direction will be the development of the source segment assembling management. This will unable different assembling scenarios and will increase the rule abstraction level and reusability.

7 References

[1] A. Bošnjaković, J. Protić, D. Bojić, I. Tartalja: Automating the Knowledge Assessment Workflow for Large Student Groups: A Development Experience, International Journal of Engineering Education, Vol. 31, No. 4, 2015, pp. 1058 − 1070.

[2] H. Geerlings, W. J. van der Linden, C. A. W. Glas: Optimal Test Design With Rule-Based Item Generation, Applied Psychological Measurement, Vol. 37, No. 2, 2013, pp. 140 − 161.

[3] M. Mišić, M. Lazić, J. Protić: A Software Tool that Helps Teachers in Handling, Processing and Understanding the Results of Massive Exams, 5th Balkan Conference in Informatics, Novi Sad, Serbia, September, 2012, pp. 259 – 262.

[4] M. Vujošević-Janičić, M. Nikolić, D. Tošić, V. Kuncak: Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments, Information and Software Technology, Vol. 55, No. 6, 2013, pp. 1004 – 1016.

[5] Đ. Pešić, S. Purić, M. Mišić, J. Protić: Software Generation of the Questions and Answers About Algorithmic Time Complexity for the Tests in Programming Courses, XXII Skup

Trendovi razvoja: Nove tehnologije u nastavi, Zlatibor, Serbia, February, 2016, pp. 53 − 56. (In Serbian).

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau: Extensible Markup Language (XML) 1.0 (Fifth Edition),

Available at: https://www.w3.org/TR/xml/

[7] Đ. Pešić, S. Purić, M. Mišić, J. Protić: Software Assembly of Program Segments Based on XML-Modeled Strategies, ETRAN 2016, Zlatibor, Serbia, June 2016, pp. RT5.3.1 − 6. (In Serbian).

[8] R. Sedgewick, P. Flajolet: An Introduction to the Analysis of Algorithms, 2nd Edition, Addison-Wesley, Boston, 2013.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, 3th Edition, The MIT Press, Cambridge, 2001.

[10] A. M. Turing: On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Vol. 42, No.1, 1937, pp. 230 − 265.

[11] A. W. Biermann: A Simple Methodology for Studying Program Time Complexity, Computer Science Education, Vol. 1, No. 4, 1990, pp. 281 − 292.

[12] M. L. Scott: Programming Language Pragmatics, Morgan Kaufmann Publishers Inc., San Francisco, 2000.

[13] Đ. Pešić, M. Mišić, J. Protić, M. V. Janicic: The Program Segment Assembling System for Examination in the Field of Algorithmic Time Complexity, ETRAN 2017, Kladovo, June 2017, pp. RT3.1.1 – 6. (In Serbian).

[14] S. Fitzpatrick: Pascal EBNF Definition,

Available at: http://www.fit.vutbr.cz/study/courses/APR/public/ebnf.html

[15] E. Ort, B. Mehta: Java Architecture for XML Binding (JAXB),

Available at:

http://www.oracle.com/technetwork/articles/javase/index-140168.html

[16] P. Terence: The Definitive ANTLR 4 reference, Pragmatic Bookshelf, Dallas, 2013.

[17] A. Z. Pinkus, S. Winitzki: Yacas: A Do-it-Yourself Symbolic Algebra Environment, Artificial Intelligence, Automated Reasoning, and Symbolic Computation, Springer, Berlin, 2002, pp. 332 − 336.

[18] B. Anders, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, J. Simeoun, Xml path language (xpath). World Wide Web Consortium (W3C) (2003),

Available at: http://www.w3pdf.com/W3cSpec/XPath/1/xpath20.pdf

[19] JavaFx,

Available at: http:.//docs.oracle.com/javafx/2/overview/jfxpub-overview.htm