Journal of Theoretical and Applied Information Technology
15th March 2019. Vol.97. No 5
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

EXTENDING THE TRANSLATION FROM PSEUDOCODE TO SOURCE CODE WITH REUSABILITY

[1]SHEILA NURUL HUDA, [2]*ZAINUDIN ZUKHRI, [3]TEDUH DIRGAHAYU, [4]CHANIFAH INDAH RATNASARI

[1,2,3,4] Department of Informatics, Universitas Islam Indonesia, Yogyakarta, Indonesia

E-mail: [1]sheila@uii.ac.id, [2]zainudin@uii.ac.id, [3]teduh.dirgahayu@uii.ac.id, [4]chanifah.indah@uii.ac.id
* Corresponding author

ABSTRACT

Pseudocode is made of a set of words in a natural language and a set of conventions to define algorithms. Pseudocode is written in a natural language that is convenience for students. In our previous work, we have developed an automatic translation from pseudocode into source code. In this paper, we extend it to handle constructs which have not been covered yet, including two types of iteration, functions, and procedures call to complete all constructs for defining algorithms. Our translation approach uses an intermediate model in XML that benefits us with the reusability of translation modules. Using reusability, we develop a new translation from pseudocode in English to source code in C++. Pseudocode and the corresponding intermediate model are Platform-independent Models (PIMs). This allows us to translate them to source code in different programming languages. The source code resulted from the translation process is Platform-Specific Model (PSM). In the translation process, it must be ensured that all models, i.e. pseudocode, the corresponding intermediate model, and the resulted source code, represent the same algorithm. Therefore, we define a conceptual metamodel for defining all the models. This paper contributes a new approach that allows reusability based on a conceptual metamodel for preserving the behavioral equivalence between all the models.

Keywords: *Pseudocode, Source Code, Conceptual Metamodel, Translation, Reusability*

1. INTRODUCTION

In learning algorithms, there are two types of students, i.e. (i) students who prefer to use a graphical method, e.g. flowchart, and (ii) students who prefer to use a verbal method, e.g. pseudocode [1]. In spite of this fact, introductory textbooks in computer science mostly present algorithms verbally using pseudocode. An algorithm defines explicitly what must be done by a computer. The algorithm is later implemented in a chosen programming language as a computer program.

Pseudocode is made of a set of words in a natural language and a set of conventions to define algorithms [2]. Pseudocode is usually written in a natural language that is convenience for the students. For example, pseudocode in most textbooks in Indonesia uses Bahasa Indonesia. The use of a natural language, instead of an unfamiliar programming language, is intended to make easier for the students in understanding algorithms [3].

The widespread use of computer programs in various fields other than computer science, such as biology, physics and medicine, increases the need to use pseudocode. Experts in those fields have to communicate their ideas with programmers in order to develop computer programs to facilitate their works. The programmers should not expect that the experts are familiar with programming languages. Hence, in most common situations, both parties have to communicate in a natural language using pseudocode.

In [4], it was concluded that less natural and complex syntax in programming languages is a major obstacle for novice students. It can also be a major barrier to understand computational thinking. Computational thinking emphasizes students' ability to think at the level of abstraction to solve problems [5]. The ability to think at this level of abstraction is important for a computer scientist in designing computer programming algorithms.

Other research on translation from pseudo code to a target programming language had been carried out. The research in [7] simplifies and structures pseudocode using XML. This approach requires pseudocode to have a strict structure. However, when we consider that, in general, novice students who are beginning to learn algorithms and programming do not yet know and learn XML, the use of XML would be a drawback.

In our previous work [6], we have developed an automatic translation from pseudocode to source code. In this paper, we extend the translation to handle other constructs that have not been handled yet, i.e. iteration, function, and procedure call to complete all constructs that had been developed previously.

This paper is presented in the following structure. Section 2 discusses the approach we use. Section 3 discusses the details of the pseudocode structure and the intermediate model in XML format for some constructs. Section 4 illustrates the use of our translation in a case study. Section 5 indicates the conclusions and subsequent work of our research.

2. REUSABILITY IN TRANSLATION WITH A CONCEPTUAL METAMODEL

In our approach, we use pseudocode written in a natural language, i.e. Bahasa Indonesia, that follows a certain format [8] so that it is easier to understand by students. XML is used as an intermediate format to allow reusability in the development of translations from pseudocode in different natural languages, i.e. not only in Bahasa Indonesia, to source code in different programming languages, i.e. not only in C++ as in our original translation. This intermediate format [9][10] allows us to decompose the translation into several smaller translation modules. These translation modules can be reused in other translations.

In this context, reusability is a quality that indicates the ability to reuse existing translation modules in the development of a new translation. Our translation consists of two translation modules that are (i) a module to translate pseudocode in Bahasa Indonesia to an intermediate format in XML and (ii) a module to translate the intermediate format in XML to source code in C++. When one need to develop a new translation from pseudocode in English to source code in C++, one can develop a module to translate from the pseudocode to an intermediate model and reuse the existing module that translates the intermediate model to source code in C++. Moreover, the development and maintenance of smaller translation modules would be more efficient than large monolithic translations.

Three types of models are involved in our translation approach, i.e. pseudocode, intermediate model, and source code. They represent algorithms in different syntaxes, i.e. in a natural language, XML format, and a programming language, respectively. To ensure the equivalence of the semantics between those models, we define a conceptual metamodel as an underlying common model from which all the models can be represented or derived [6]. The conceptual metamodel is shown in Figure 4. The metamodel is defined by considering the grammar of common programming languages.

Our translation approach considers pseudocode as a model or representation of an algorithm. Pseudocode is taken as an input that will be translated into target artifacts, such as source code in programming languages. The approach uses an intermediate model in XML format so as to introduce reusability to the translation modules. For example, we can develop translations from different natural languages or to different programming languages. Pseudocode and the intermediate model are Platform-Independent Models (PIMs); whereas the output, i.e. source code in a target programming language, is Platform-Specific Model (PSM).

By using an intermediate model that is platform independent, further development of different translations can benefit from the reusability because the process of translation from a pseudocode to an intermediate model is decoupled from the process of translation from the intermediate model to source code in a target programming language. For example, we can develop translation of pseudocode in English.

Our approach is validated in two aspects, i.e. syntax and semantic. For the syntax aspect, we define an XML schema as an implementation of the conceptual metamodel. Using this XML schema, we can validate whether the intermediate model complies with the conceptual metamodel. The syntax of the source code is validated by compiling the resulted source code. A successful compilation indicates that the syntax of the source code is valid and hence comply with the conceptual metamodel.

For the semantic aspect, the validation is done by running the executable program resulted from the source code compilation. When the program execution behaves as the algorithm defined in the pseudocode, it indicates that the semantic is valid.

While the validation of the syntax aspect can be done automatically using available tools, the validation of the semantic aspect has to be done manually. For complex algorithms, this way of validation requires intense work and is prone to human error.

3. PSEUDOCODE AND XML STRUCTURE

In this section, we discuss the structure of pseudocode that we adopt and the structure of the intermediate model that we develop.

Pseudocode describes an algorithm using a set of words in a natural language and a set of loose conventions. The fact that there are a lot of token variations in a natural language is a challenging issue. Therefore, in our translation, we restrict the structure of pseudocode by adopting a popular pseudocode format used by academia in Indonesian higher education [8]. The format is then customized to accept tokens in Bahasa Indonesia and is extended to accept tokens in English.

3.1 Structure of Pseudocode

A pseudocode consists of one or more modules that can be functions, procedures, or a program. Figure 5 shows an excerpt of the grammar that implements the metamodel for the module construct.

A module consists of three sections, i.e. title, dictionary, and algorithm. A variable declaration is placed in the dictionary section. The algorithm section contains a sequence of statements. These statements can be input, output, assignment, decision, or iteration statements. The intermediate format represented in XML format for this construct is shown in Figure 6.

3.2 Decision Construct

A decision consists of one or more cases; each of which has its own condition. The order of case statements is important and hence considered in the translation process. Figure 7 shows the grammar for the decision construct. Token IF and ELSE may use different terms or words in Bahasa Indonesia with their respective meanings. Similarly, when we work with pseudocode in English, different tokens with the same meanings can also be used. This extension for pseudocode in English creates opportunity for the translation to be used in wider academia society.

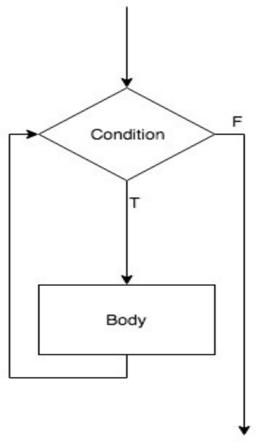
Furthermore, a decision can contain one last case without condition. This case serves as a default case when conditions in all other cases cannot be met. Each case consists of a series of actions that will be executed when the case's condition is met. The XML format for the decision construct is shown in Figure 8.

3.3 Iteration Construct

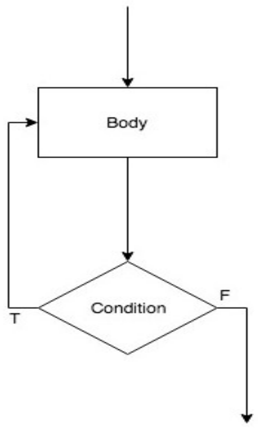
We distinguish an iteration into two forms, namely type-1 (WHILE-DO) iteration and type-2 (REPEAT-UNTIL) iteration. A type-1 iteration checks the condition first. If the condition is met, it will execute a series of statements and then return to condition checking. A type-2 iteration executes a series of actions first, then checks the condition to proceed to the next iteration. The semantics of both iteration forms can be seen in Figure 1.

Figure 9 shows the grammar for the constructs of both types of iteration. Token WHILE, DO, REPEAT, and UNTIL may use different terms or words in Bahasa Indonesia or English with respective meaning.

In an intermediate model, both forms of iterations have that same XML format, but are indicated by the *type* attribute. The XML format for the iteration constructs is shown in Figure 10.



(a) Type-1 Iteration



(b) Type-2 Iteration

Figure 1: Semantic Difference between (a) Type-1 and (b) Type-2 Iteration

4. CASE ILLUSTRATION

Our previous work was limited in terms of constructs and the natural language for defining pseudocode, i.e. Bahasa Indonesia. In this work, we apply the advantage of reusability to extend the translation from the input model, i.e. pseudocode in English, to an intermediate model. As we can reuse the translation module from the intermediate model in XML to source code in C ++, we need only to develop the module to translate pseudocode in English to an intermediate model in XML.

In this case illustration, we present a pseudocode in Bahasa Indonesian as it is a mother tongue used by the majority of students in Indonesia.

4.1 Decision

Figure 11 shows pseudocode in Bahasa Indonesia that defines a simple algorithm to determine the form of water, given an input of the water temperature. Water can be in the form of solid, liquid, or gas depending on the temperature. The translation is developed using ANTLR Parser Generator that generates a parse tree for further processing to an intermediate model in XML. The translation result is shown in Figure 12.

All the XML elements, i.e. program, variables, sequences, input, outputs, decision, and cases, in the resulted intermediate model comply with an XML schema that are defined as an implementation of the conceptual metamodel. This compliance preserves the equivalences of algorithm behavior as in the corresponding pseudocode.

This intermediate model is then translated to source code in C++ using XSLT. The source code resulted from the translation is shown in Figure 2. This source code complies with the C++ grammar which also complies with the conceptual metamodel. This compliance ensures that the source code has an equivalent behavior with the corresponding pseudocode and intermediate model.

4.2 Iteration

Figure 13 shows pseudocode in English that defines a simple algorithm that repeatedly receives inputs as temperature until the input entered is less than 100 ^{O}C. When this condition is met, the algorithm will conduct second iteration to repeatedly receive inputs as temperature until the input entered is higher than 100 ^{O}C. This pseudocode contains both types of iteration. The first iteration is a type-1 iteration and the second is a type-2 iteration.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  double t;
  cin  >> t;
  if (if (t < 0) {
    cout  << "bentuk padat";
  }
  else if (t < 100) {
    cout  << "bentuk cair";
  }
  else {
    cout  << "bentuk gas";
  }
}
```

Figure 2: Source Code with Decision Case as the Output

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  double t;
  t  = 200;
  while (t < 100) {
    cin  >> t;
  }
  do {
    cin  >> t;
  } while (T > 100);
}
```

Figure 3: Source Code with Iteration Case as the Output

The translation results in an intermediate model in XML shown in Figure 14. Iteration type and condition are represented as attributes of the *iteration* element. This result complies with the defined the XML schema of the conceptual metamodel.

The translation results of the intermediate model to source code in C ++ is shown in Figure 3. A type-1 iteration is translated to a while-do construct and a type-2 iteration is translated to a do-while construct.

5. CONCLUSION

In this paper, the development of automatic translations from pseudocode in natural languages, i.e. Bahasa Indonesia and English, to source code in a target programming language, i.e. C++, has been carried out. The translation uses an intermediate model in XML to decouple between (i) the translation from pseudocode to an intermediate model and (ii) the translation from the intermediate model to source code. This decoupling is to provide reusability in the development of new translations. The development of a translation module that translates pseudocode in English to an intermediate model is the utilization of this reusability.

To ensure the behavioral equivalence between the models involved, i.e. pseudocode, intermediate model, and source code, a conceptual metamodel is defined to represent the behavior of all the models.

ACKNOWLEDGEMENT

This work is part of the project sponsored by the Directorate of Research and Community Services of the Indonesian Ministry of Research, Technology, and Higher Education, 2018.

REFERENCES:

[1] Scanlan DA. Learner preference for using structured flowcharts vs. pseudocode when comprehending short, relatively complex algorithms: A summary analysis. Journal of Systems and Software. 1988; 8(2): 145-155.

[2] Robertson LA. Pseudocode. Encyclopedia of Information Systems. 2003: 575-588.

[3] Oda Y, Fudaba H, Neubig G, Hata H, Sakti S, Toda T, and Nakamura S. Learning to generate pseudo-code from source code using statistical machine translation. Proc. 30th IEEE/ACM Intl. Conf. Automated Software Engineering. 2015; pp. 574-584.

[4] Kelleher C, and Pausch R. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. ACM Computing Surveys. 2005; 37 (2) 83–137.

[5] Wing J. CT. Communications of the ACM. 2006; 49 (3) pp. 33-35.

[6] Dirgahayu T, Huda SN, Zukhri Z, Ratnasari CI. Automatic translation from pseudocode to source code: A conceptual-metamodel approach. 2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom). Phuket. 2017: 122-128.

[7] Mukherjee S, and Chakrabarti T. Automatic algorithm specification to source code translation. Indian J. Computer Science and Engineering (IJCSE). 2011; 2 (2) pp. 146-159.

[8] Liem I. Diktat Algoritma dan Pemrograman: Pemrograman Prosedural. Bandung: STEI ITB. 2007.

[9] Parekh V, and Nilesh D. Pseudocode to source code translation. Intl. J. Emerging Technologies and Innovative Research (JETIR). 2016; 3 (11) pp. 45-52.

[10] Dirgahayu T, Quartel D, and van Sinderen M. Development of transformations from business process models to implementations by reuse. Proc. 3rd Intl. Workshop on Model-Driven Enterprise Information Systems (MDEIS). 2007: pp. 41-50.

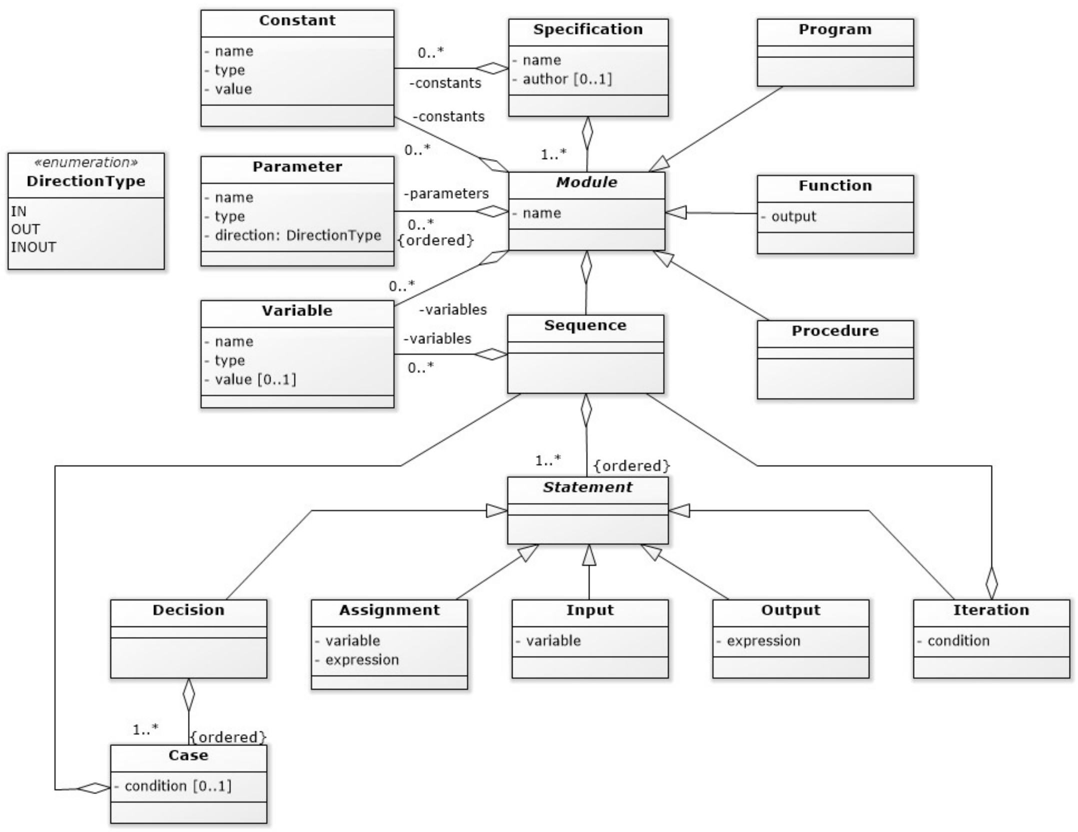


Figure 4: Conceptual Metamodel

```
grammar pseudocode_eng;
spec: module+ EOF;
module: title dictionary algorithm;
algorithm: description?  sequence;
sequence : (statement | comment)*;
statement : input | output | assignment | decision| iteration |procedure_call;
```
Figure 5: Grammar for The Module Construct

```
<?xml version="1.0" encoding="UTF-8"?>
<specification xmlns="informatics.uii.ac.id/pseudo/1.0" name="example">
   <program name="main">
      <variables>
      .
      .
      .
      </variables>
      <sequence>
      .
      .
      .
      </sequence>
   </program>
   <function name="example1" output="output_type">
      <parameter name="par_name" type="par_type"/>
      .
      .
      .
      <parameter name="par_name1" type="par_type1"/>
         <variables>
         .
         .
         .
         </variables>
         <sequence>
         .
         .
         .
         </sequence>
   </function>
   <procedure name="example2">
      <parameter name="par_name" type="par_type" direction="in/out"/>
      .
      .
      .
      <parameter name="par_name1" type="par_type1" direction="in/out"/>
      <variables>
      .
      .
      .
      </variables>
      <sequence>
      .
      .
      .
      </sequence>
   </procedure>
</specification>
```

Figure 6: XML Format for The Module Construct

```
decision: IF condition openthen sequence closethen (ELSE (elseif | other))?;
condition: expression;
elseif: decision;
other: '(' sequence ')';
```

Figure 7: Grammar for The Decision Construct

```
<decision>
   <case condition="some_expression">
      <sequence>
      .
      .
      .
      </sequence>
   </case>
   <case condition="some_expression">
      <sequence>
      .
      .
      .
      </sequence>
   </case>
   <case>
      <sequence>
      .
      .
      .
     </sequence>
   </case>
</decision>
```

Figure 8: XML Format for The Decision Construct

```
iteration1: WHILE condition DO open_itr sequence close_itr;
iteration2: REPEAT open_itr sequence close_itr UNTIL condition;
condition: expression;
```

Figure 9: Grammar for The Iteration Construct

```
<iteration type = "1" condition = "some_condition">
   <sequence>
   .
   .
   .
   </sequence>
</iteration>
<iteration type = "2" condition = "some_expression">
   <sequence>
   .
   .
   .
   </sequence>
</iteration>
```

Figure 10. XML Format for The Iteration Construct

```
Program Menentukan_bentuk_air
{Menentukan bentuk air jika diketahui suhu air dalam °C. Menggunakan statement
if-then-else}
Kamus
T : real {suhu dalam °C}
Deskripsi Algoritme
baca(T)
jika (T kurang dari 0) maka (
  tulis("bentuk padat")
)
selain itu
jika (T kurang dari 100) maka (
  tulis("bentuk cair")
)
selain itu (
  tulis("bentuk gas")
)
```

Figure 11: Example of Pseudocode with Decision Cases

```
<?xml version="1.0"  encoding="UTF-8"?>
<specification xmlns= "informatics.uii.ac.id/pseudo/1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
name="Menentukan_bentuk_air">
   <program name="main">
      <variables>
         <variable name="t" type="double"/>
      </variables>
      <sequence>
         <input variable="t"/>
         <decision>
            <case condition="t &lt; 0">
               <sequence>
                  <output expression="&quot;bentuk padat&quot;"/>
               </sequence>
            </case>
            <case condition="t &lt; 100">
               <sequence>
                  <output expression="&quot;bentuk cair&quot;"/>
               </sequence>
            </case>
            <case>
               <sequence>
                  <output expression="&quot;bentuk gas&quot;"/>
               </sequence>
            </case>
         </decision>
      </sequence>
   </program>
</specification>
```

Figure 12: Intermediate Model in XML with Decision Cases

```
Program Receiving_input_iteratively
{Receive input of temperature repeatedly until certain condition }
Dictionary
T : real {temperature in °C}
Algorithm Description
T is 200
while T is less than 100 do (read(T))
repeat (read(T)) until T>100
```

Figure 13: Example of Pseudocode with Iteration

```
<?xml version="1.0" encoding="UTF-8"?>
<specification xmlns="informatics.uii.ac.id/pseudo/1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
name="Menerima_input_iteratif ">
   <program name="main">
      <variables>
         <variable name="t" type="double"/>
      </variables>
      <sequence>
         <assignment variable="t" expression="200"/>
         <iteration type="1" condition="t &lt; 100">
            <sequence>
               <input variable="t"/>
            </sequence>
         </iteration>
         <iteration type="2" condition="t &lt; 100">
            <sequence>
               <input variable="t"/>
            </sequence>
         </iteration>
      </sequence>
   </program>
</specification>
```

Figure 14: Intermediate Model in XML with Iteration