AUTOMATIC ALGORITHM SPECIFICATION TO SOURCE CODE TRANSLATION

SUVAM MUKHERJEE

Department of Computer Science and Engineering,
Institute of Engineering and Management,
Y-12, Block EP, Sector V
Salt Lake, Kolkata 700091, India.

TAMAL CHAKRABARTI

Assistant Professor
Department of Computer Science and Engineering,
Institute of Engineering and Management,
Y-12, Block EP, Sector V
Salt Lake, Kolkata 700091, India.

Abstract

Computers have become all-pervasive, and are being used in a variety of areas like Microbiology, Astronomy, Social Sciences and many others. In almost all these areas, algorithmic solutions to problems are common. However, most programming languages have certain idiosyncrasies. This is why people who don't have a good background in computer programming have difficulty in writing good, efficient programs. Moreover, there are many programming languages which allow coding in a variety of paradigms. Though it is easy for someone trained in Computer Science to convert a program from one language to another, it is less so for people in other fields. In this paper, we describe a translation program that can create a piece of executable code, given the code's algorithmic specification. This program allows the user to specify his/her code using an easy-to-understand, simple-to-write and more or less immutable pseudo code specification. The program will then check the pseudo code for errors, and convert it to a specified language (be it C, Java, or any other language). The program may easily be extended to accommodate different languages. Our program allows the user to focus on just the algorithm, and not on issues related to implementation.

Keywords: Pseudo code implementation; Code Generation; Regular Expressions; Pattern Matching; Hash Tables; XML to C, Java;

1. Introduction

Of late, Algorithms have been applied in diverse areas like Microbiology, Astronomy, Social Sciences and many others. Today, more and more problems are lending themselves to algorithmic solutions. In Numerical Methods, we have algorithms for solving equations by iteration [7], solving simultaneous algebraic equations [7], doing interpolation [1], performing differentiation and integration [1] and so forth. In Graph Theory, we have algorithms like building minimum spanning trees using Kruskal's algorithm [4]. Graphs are used in a variety of web based applications like Facebook and MySpace, and many of the operations in these applications are nothing but implementations of graph theoretic algorithms. Many algorithms are being devised to tackle biological problems like aligning DNA sequences [2].

Algorithms, however, need to be implemented; and this is where a problem lies. People who don't have a good background in computer programming have difficulty in writing good, efficient programs. Moreover, there are many programming languages which allow coding in a variety of paradigms. So it is not easy for someone not trained in Computer Science to convert a program from one language to another.

This paper describes an innovative translation process that can create a piece of executable code, given the code's algorithmic specification. The specification of the algorithm is done in XML. In the specification, the user need not bother about any implementation details. This allows users to write their algorithms without having to write the computer programs that would implement the algorithms.

Although the imperative programming language ALGOL became one of the standard ways in which to describe algorithms, our formal framework for writing pseudo codes is much more intuitive and provides a higher level of abstraction. This has been illustrated later in the paper by taking a simple example. Moreover, there have been translation programs designed for ALGOL, like the MARST [5], Whetstone KDF9 ALGOL Translator [8] and the ALGOL 60 Translator for X1 [3]. However, none of these attempt to allow translation from the pseudo code specification to *any* other language. This is what our translation process achieves. Our implementation of the translator, called ALGOSmart, can presently translate pseudo code specification written in XML into C and

Java; and later in this paper, we have discussed why our translation process can be easily extended to accommodate other programming languages.

The rest of the paper is organized as follows: Section 2 describes the basic operation of the entire program, Section 3 takes a closer look at the operation of the translator, and Section 4 contains some demonstrations. Finally, Section 5 gives a Conclusion.

2. The Basic Operation

To illustrate the basic operation with a small example, let us assume that the user wishes to enter a value from the keyboard, and store it in a variable named "a". Let us further assume that "a" can store integral values. Then the code which will achieve this in C is:

```
scanf("%d", &a);
```

Again, doing the same thing in Java would require the following statements:

```
BufferedReader br =new BufferedReader(newInputStreamReader(System.in));
String temp;
temp = br.readLine();
int a = Integer.parseInt(temp);
```

Note that both the codes achieve the same basic objective- and yet there are details that need to be remembered which may be entirely beyond the scope of the user's work. What the user really wants to do is write this line:

```
<vwrite> a </vwrite>
```

and then expect, somehow, this line to get converted into the correct statement (in either C or Java). This is precisely what the translator does.

The primary mechanism used by the translator is pattern matching [6] using regular expressions. Regular expressions offer a compact representation of a particular pattern. The translator makes a single pass over the entire pseudo code, and it checks for known patterns. Once a known pattern is found, it tries to understand the context, and then take a particular course of action.

The single pass feature of the translator makes it fast- its time complexity is linear in the number of lines that are present in the implementation.

The translator, during the translation process, builds some databases. These databases are all Hash Tables. Hashes are used because of its fast access time- and this access time stays fast no matter how many elements one puts into the hash table [10].

The following section demonstrates a black box specification of the entire program.

2.1. *Black Box Specification*

Our translator has two primary modules. The inputs and outputs of each module, and their interrelation are shown in the following figure:

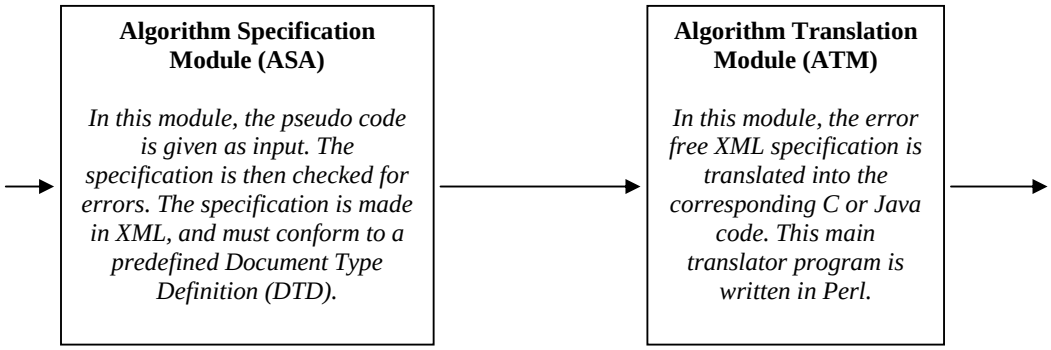

Fig 1: Black box specification of the entire program

2.1.1 *Algorithm Specification Module (ASM)*

The purpose of this module is to provide the user with a formal framework for writing pseudo codes. In this module, the specification is written in XML. In XML, there are no predefined tags; and this allows us to predefine a set of tags, along with their structures. This definition is called the **Document Type Definition (DTD)**.

The elements within a XML document forms a document tree [11]. An XML document having the correct syntax is said to be *well-formed*. When the XML document, in addition to being well-formed, also conforms to the predefined DTD, then it is called *valid*. The basic rules for an XML document to be syntactically correct are [12]:

1. XML documents must have a root element
2. XML elements must have a closing tag
3. XML tags are case sensitive
4. XML elements must be properly nested
5. XML attribute values must be quoted.

The Document Type Definition (DTD) simply defines a set of tags, and the way the tags may be structured. For example, in our program, every algorithm must begin with a **start** tag. Within the start tag, we may have zero or more **function**s, and there has to exactly one **main**. This is enforced by writing the following line in the DTD:

```
<! DOCTYPE start
[
<!ELEMENT start (function*, main)>
]>
```

We observe that the definition itself contains a regular expression- function*- which indicates that 0 or more functions may be present in the specification. The "," indicates that the **main** must come after all function definitions. Lastly, the expression **main** implies that there can be one, and exactly one, occurrence of **main** in the specification.

Of course, the user need not bother about what a DTD is, or what the DTD for this program is. All the user needs to know is the fact that he/she can write as many functions as he/she desires, and there must be a **main** routine.

The following is a list of tags that the user is allowed to use, along with the way each tag may be used. Note that the following table is all that the user needs to know in order to start writing algorithms.

Tag	Operation
<start> … </start>	The root tag. This tag *must* be present, and the entire specification should be encapsulated within these two tags.
<var> variable_name = x </var>	If x is an Integer, it defines an integer variable with name *variable_name* and initializes it to value *x*. If x is a Real number, it defines a double variable with name *variable_name* and initializes it to value *x*.
<vread> variable_name </vread>	Read the value of variable named *variable_name* from the keyboard.
<vwrite> variable_name </vwrite>	Write the value of the variable named *variable_name* to the monitor.
<array> integer array_name[x] </array>	Define an array of integer values. The range of the array goes from 0 to (*x*-1). Here *x* must be a positive integer.
<array> real array_name[x] </array>	Define an array of real values. The range of the array goes from 0 to (*x*-1). Here *x* must be a positive integer.
<aread> array_name[i] </aread>	Read the value of element *i* of the array named *array_name* from the keyboard.
<awrite> array_name[i] </awrite>	Write the value of element *i* of the array named *array_name* to the monitor.
<write> some text </write>	Just print some text in the monitor.
<n></n>	Go to a new line.
<s> *statement* </s>	The general statement tag. Statements are assignments, function calls, conditional-statements or loops.
<s> if (condition) then </s> . . . <s> endif </s>	A basic conditional statement. The condition must be enclosed within parenthesis.
<s> if (condition) then </s> . . . <s> elseif (condition) then </s> . . . </endif> </endif>	A nested conditional statement.
<s> if (condition) then </s> . . . <s> else </s> . . . <s> endif </s>	A basic if-else block.
<s> loop from i=a to (condition) step x </s> .	This is the basic loop statement. The loop index is the variable ***i***, which is initialized immediately after the word **from**. The loop continues until the condition,

. . <s> endloop </s>	placed within parenthesis, turns out to be false. The **step** keyword indicates that ***i*** should incremented in steps of *x*, where *x* is an integer. Basically, it translates to the statement $i = i + x$. The initialization statement may be omitted. The assumption, in this case, is that the loop index has been defined previously.
<s> loop till (condition)</s> . . . <s> endloop </s>	As the name suggests, this loop is used when it is not known exactly how many times a loop will occur. In that case, the termination of the loop depends on the condition- when the condition turns out to be false, the loop halts.
<function> <header> function_name(real x, integer y, real a[], integer b[],…) returns real </header> . . . </function>	This defines a function. A function starts with the **function** tag. It is then followed by the **header**- which comprises the function name, followed by the list of arguments. The arguments may variables or arrays, and may be either real or integers. Lastly, we have the **returns** keyword, which indicates the type of value returned by the function. If the function returns a real number, as shown here, we write **returns real**. If it returns an integer, then we write **returns integer**. If the function doesn't return a value, we simply omit the **return** keyword altogether.

Table 1: The Formal Framework to specify pseudo codes

The XML document is then mapped to its corresponding tree structure. The DOM (Document Object Model) defines a standard way for accessing and manipulating documents. The XML DOM views an XML document as a tree. If we can successfully construct a DOM tree object from an XML document, then the document is error free.

To compare our framework for specifying algorithms with that of ALGOL, let us consider the simple program of printing the message "Hello World". The ALGOL version of the program is taken from the University of Michigan, Department of Computer Science Language Guide [9].

```
// the main program (this is a comment)

BEGIN
FILE F (KIND=REMOTE);
EBCDIC ARRAY E [0:11];
REPLACE E BY "HELLO WORLD!";
WHILE TRUE DO
  BEGIN
  WRITE (F, *, E);
  END;
END.
```

Fig 2: ALGOL Specification to print "HELLO WORLD!"

The same specification, using our framework, would be:

```
<start>
<main>
<write> HELLO WORLD! </write>
</main>
</start>
```

Fig 3: The "HELLO WORLD" specification using our pseudo code framework

A comparison of Fig 2 and Fig 3 reveals the simplicity and intuitive nature of our framework.

2.1.2 *Algorithm Translation Module (ATM)*

The algorithm specification is now fed into the ATM. This module is written entirely in Perl, and uses regular expressions and pattern matching extensively. In the ATM, each line of the XML document is read- one line at a time. The line is then processed to understand the operation which has been invoked in that line. As this is a *translation* module, a single pass is sufficient. We do not attempt to optimize the code, or otherwise deduce other semantic properties. These tasks are left to the compiler of the language.

The output of the ATM is a program- it is the implementation of the algorithm in the desired language. The user may even choose to directly run the program. The ATM would then invoke a system call, which in turn would create the executable file and then run it.
Now that we have described the operation of the whole program, we take a closer look at the translation process itself.

3. The Translation Process

The translation process is all about using regular expressions and matching patterns. A regular expression is a compact way of describing a pattern. We have chosen Perl to write this module because Perl is particularly well equipped to handle regular expressions. In this section, we first take a look at the idea behind the basic operation of the translation process. We give an algorithm for the translation process as a whole. Central to the translation process is how we handle individual operations- and this is what we take a look at next. We take up a sample operation- var- and take a look at how it is handled. Finally, we give a general algorithm for the operation handler.

3.1. *Basic Operation*

During the translation process, each line of the specification is read- one line at a time. There is a predefined set of patterns- each pattern representing a particular operation. The translator now tries to match the operation in the line with one of these patterns. When a match is found, the appropriate handler is invoked.
For example, the **<var>** operation would match the pattern **^\s*<var>.*$**. The pattern has been expressed in Perl. The interpretation of this regular expression is as follows: The operation is **var** if the pattern obtained starts with 0 or more spaces, followed by <**var**>, and is followed by 0 or more occurrences of any character.
Once the match is found, a variable called **$var** is set. This is a scalar variable (Perl has two other built in data types- the array, denoted by @, and hashes, denoted by %.)
We now give an algorithm of the translation process. # denotes comments.

```
TRANSLATE (line)
# Here line holds a single line from the XML document.

    1.  Match the operation in line with one of the predefined operation patterns.
    2.  If found, set the appropriate variable to indicate the operation found.
    3.  Run the corresponding operation handler.
    4.  Reset the variable- handling is complete.

TRANSLATOR (specification)
# "specification" is the XML file containing the algorithm specification

    1.  i = 0
    2.  for each line in the specification, do
    3.         begin
    4.                store the i-th line in the variable line
    5.                TRANSLATE (line)      # Call the TRANSLATE function
    6.                i = i + 1
    7.   end for
```

Fig 4: The algorithm for the translation process

3.2. *A Sample Operation Handling Process*

In Fig 4, step 3 of the **Translate** function is of importance. Here we take up an operation and examine how it's handled. The other operations are handled in a similar way. We take up the **<var>** operation because it also demonstrates how we create and handle the symbol table.

```
VAR_HANDLER
# Received a statement like <var> a = 0 </var>. symtab is the symbol table.
    1.  Replace the <var> and </var> with blank spaces.
    2.  # The statement now becomes a = 0
    3.  Extract the variable name and the initialization value. Store them in the
        variables name and number respectively.
    4.  # Now name contains a, and number contains 0.
    5.  Remove all leading and trailing spaces from name and number.
    6.  # This is crucial because we will use the name to hash into the hash table.
    7.  Check the type of initialization value used in number. If the initialization
        value has a pattern \d+, which means 1 or more digits, then it is an integer. If
        the pattern is \d+\.+d*, which means 1 or more digits, followed by a decimal
        point, followed by 0 or more digits- then it is a real number. Store the obtained
        type in the variable type.
    8.  If type is an integer, then insert into symtab the entry <a, integer>. Else make
        the entry <a, double>.
    9.  If type is an integer, output the definition int a = 0;
        Else output the definition double a = 0;
    10. Reset the var variable to indicate that processing is complete.
```

Fig 5: The algorithm to handle the <var> operation.

Basically, step 9 is the language dependant portion. This is where we actually worry about the syntax of the target source language. This demonstrates the measure of language independence that the translation process provides- almost the entire processing is independent of the target source language except for the final few steps. Next, we provide a general algorithm for the operation handling process.

3.3 *General Algorithm for Operation Handler*

Here, we provide an algorithm for the general operation handler in a flowchart format.

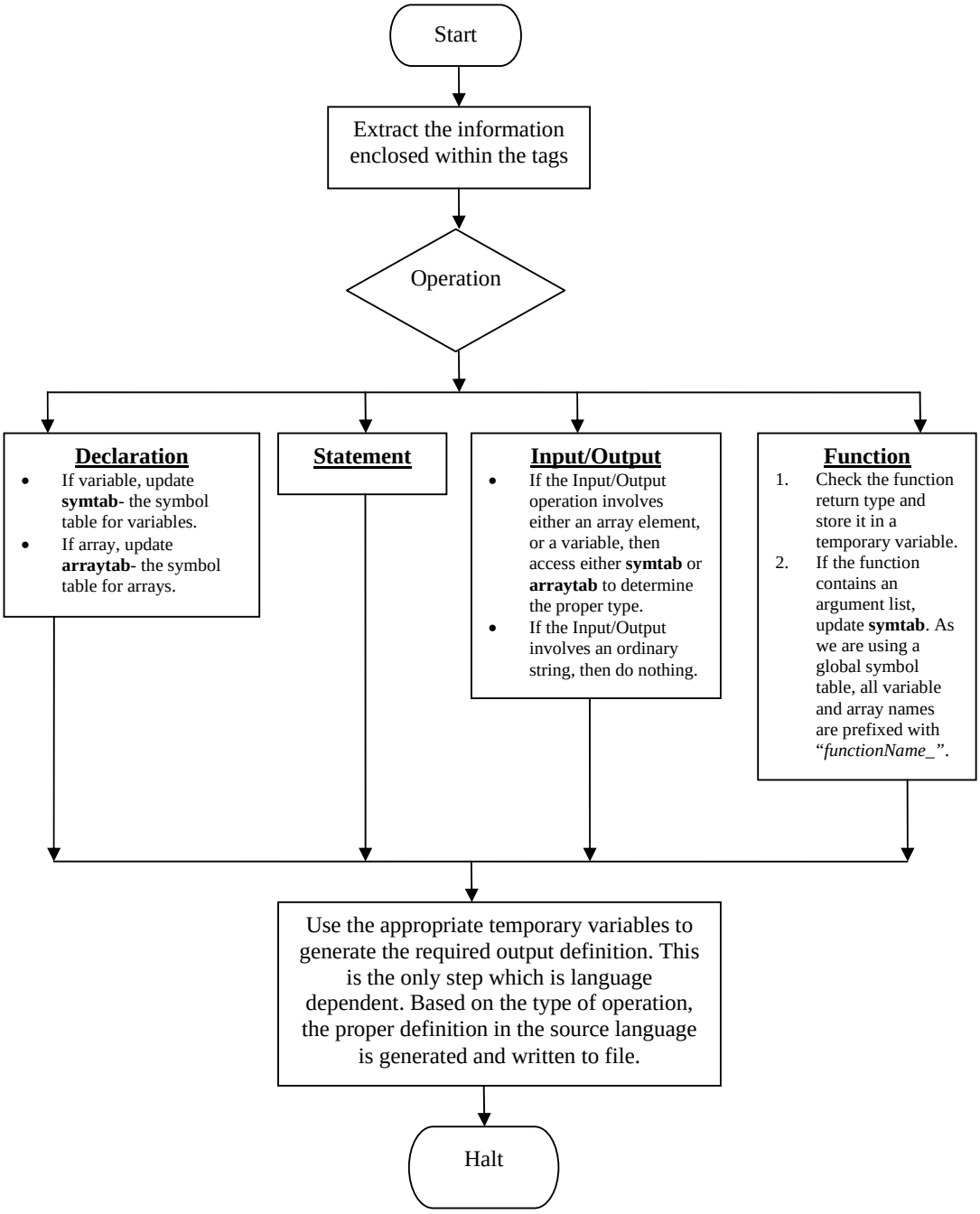

Fig 6: The flowchart describing the algorithm for a general operation handling process. This basically elaborates **Step 3** of the **Translate** function given in Fig 4

4. Demonstration

In this section, we put our translation process to test. We take up an algorithm, write the pseudo code conforming to our formal framework, and then take a look at how the translation process performs.
Our implementation of the translator is called **ALGOSmart**, and has been written in Perl.
The pseudo code has to be written in XML. Note that writing opening and closing tags may be a bit cumbersome. To alleviate this problem, the user may use a program like NetBeans, where the corresponding closing tag is automatically placed by the software, and writing pseudo codes become very easy.

4.1. *Finding roots of a polynomial equation by Successive Bisection method.*

Successive Bisection method is an iterative method to compute the roots of a polynomial equation $f(x) = 0$. The following version of the algorithm is taken from [7], and one may find a more elaborate discussion of the algorithm, including an analysis of errors, over there.
In brief, the algorithm works as follows:
We pick two trial points x_0 and x_1, such that they enclose the root. The two points enclose a root if $f(x_0)$ and $f(x_1)$ are of opposite signs. Next, we bisect the interval (x_0, x_1) and denote the mid-point by x_2. In other words, we compute:

$$x_2 = (x_1 + x_0) / 2$$

If $f(x_2) = 0$, then we have a root. However, if $f(x_2) > 0$ [see Fig 7], then the root is between x_0 and x_2. We now replace x_1 by x_2 and search for the root in this new interval, which is half the previous interval. We again calculate $f(x_2)$ at the midpoint of this new interval. Let $f(x_2) < 0$ this time. Thus, the root lies between x_2 and x_1. We now replace x_0 by x_2 and again bisect the interval. This is how the algorithm proceeds towards the root.

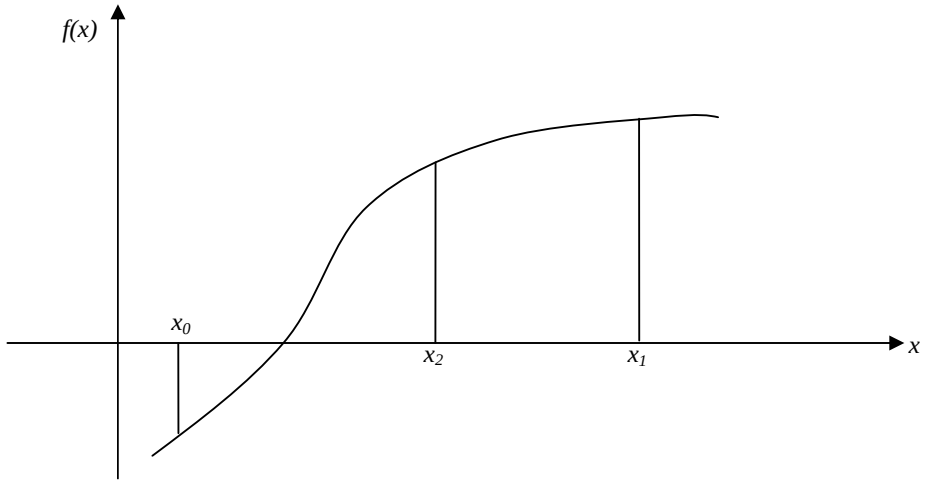

Fig 7: The Successive Bisection Method

We now present the exact Successive Bisection Algorithm given in [7], written as an XML document and conforming to our Document Type Definition. An inspection will show that the XML specification is not much different from the one given in [7]. We will use this pseudo code to convert it to both C and Java programs. We then test it using the same equation considered in [7]: $x^2 – d = f(x) = 0$. We also take the same trial points as [7], $x_0 = 2.0$ and $x_1 = 7.0$, $d = 25$. We are basically computing the square root of 25, and our answer should be something around 5.

```
 1. <start>
 2.
 3. <function>
 4. <header> absolute(real x) returns real </header>
 5. <s> if (x GE 0) then </s>
 6. <s> return x </s>
 7. <s> else </s>
 8. <s> return (-1 * x) </s>
 9. <s> endif </s>
10. </function>
11.
12. <function>
13. <header> sign(real x) returns integer </header>
14. <s> if (x GE 0) then </s>
15. <s> return 1 </s>
16. <s> else </s>
17. <s> return 0 </s>
18. <s> endif </s>
19. </function>
20.
21. <function>
22. <header> f(real x) returns real </header>
23. <s> return (x*x - 25) </s>
24. </function>
25.
26. <main>
27. <var> x0 = 0.0 </var>
28. <var> x1 = 0.0 </var>
29. <var> x2 = 0.0 </var>
30. <var> f0 = 0.0 </var>
31. <var> f1 = 0.0 </var>
32. <var> f2 = 0.0 </var>
33. <var> i = 0 </var>
34. <write> Enter x0: </write>
35. <vread> x0 </vread>
36. <n></n>
37.
38. <write> Enter x1: </write>
39. <vread> x1 </vread>
40. <n></n>
41.
42. <s> f0 = f(x0) </s>
43. <s> f1 = f(x1) </s>
44.
45. <s> if (sign(f0) EQ sign(f1)) then </s>
46. <write> Starting values are unsuitable </write>
48. <s> else </s>
49. <s>loop till (absolute((x1-x0)/x1) GT 0.001)
50. <s>       x2 = (x0 + x1)/2 </s>
51. <s>       f2 = f(x2) </s>
52. <s>              i = i + 1 </s>
53. <s>       if (sign(f0) EQ sign(f2)) then </s>
54. <s>              x0 = x2 </s>
55. <s>       else </s>
56. <s>              x1 = x2 </s>
57. <s>       endif </s>
58. <s>endloop </s>
59. <s> endif </s>
60. <n></n>
61. <write> Solution converges to a root: </write>
62. <n></n>
63. <write> Number of iterations = </write>
64. <vwrite> i </vwrite>
65. <n></n>
66. <write> Root is: </write>
67. <vwrite> x2 </vwrite>
68. </main>
69. </start>
```

Fig 8: Successive Bisection Method pseudo code. The line numbers are ***not*** necessary, and have been given to ease our discussion.

We now use ALGOSmart to translate this XML pseudo code into C and Java programs. We then take up one or more operations from the pseudo code and observe how they are handled.

We start off with the **<main>** tag in line 26. This indicates the point where execution begins. **<main>** has a set of variable definitions at the beginning (lines 27 to 33). It is good practice to define all the variables at the beginning of a function. Let us take a look at the corresponding C output for these lines:

```
double x0 = 0.0;
double x1 = 0.0;
double x2 = 0.0;
double f0 = 0.0;
double f1 = 0.0;
double f2 = 0.0;
int i = 0;
```

Note that ALGOSmart has automatically detected the *type* of these numbers- thus declaring them as **double** or **int** variables as appropriate. If we initialize the values to *any* real number, the translator would declare them as double. If the initialization value is *any* integer, then the translator would declare the variable as int. This illustrates Fig 3- the translator extracts the initialization value, and tries to match it with the patterns for real or integral numbers. It then updates the symbol table and outputs the definition.

Next, we have a set of Input/Output operations. Let us look at the output, for lines 34 to 36, of ALGOSmart when the output source language is C:

```
printf("Enter x0:");
scanf("%lf", &x0);
printf("\n");
```

The translation process follows the flowchart in Fig 4. It finds that each line belongs to Input/Output category. The first line is simply printing some text, so nothing special is done- and it straightaway outputs the proper definition. Line 35, however, involves a variable. So it extracts the variable name and searches up the symbol table for an entry with this name. From the entry, the translator gets to know about the type- and this gets stored in a temporary variable. This temporary variable is then used to output the appropriate definition. In this case, the temporary variable is crucial to determine the format specifier. Double variables have the specifier %lf, while for Integers it is %d. Note that the user need not be aware of any of these details.

Let's look at a similar output, for lines 38 to 40, of ALGOSmart when the output source language is Java. At the very beginning, ALGOSmart makes the following definition:

```
static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
static String temp;
```

The **BufferedReader** object **br** will now be used to read from the keyboard. As we may have input statements in **main()**, and since in Java the **main()** is defined to be **static**, this definition is also static. So we may use **br**, even though no objects of the class exist. Using **br**, the lines 38 to 40 become:

```
System.out.println("Enter x0:");
temp = br.readLine();
x0 = Double.parseDouble(temp);
System.out.println("\n");
```

Constructs like loops and conditionals have pretty much the same structure in both C and Java. To give an idea of what these constructs look like when translated, we take a look at the Java output for lines 49 to 58:

```
for(;(absolute((x1-x0)/x1) > 0.001);)
 {
              x2 = (x0 + x1)/2 ;
              f2 = f(x2) ;
              i = i + 1 ;
              if (sign(f0) == sign(f2)) {
                     x0 = x2 ;

}
```

We now come to functions. This pseudo code uses three functions- *absolute*, *sign* and *f*. All three functions take arguments and return a result. We first get an idea of how *absolute* looks like after it has been translated into C code:

```
double absolute(double x)
{
 if (x >= 0) {
 return x ;

}
else
{
 return (-1 * x) ;
 }


}
```

When the translator encounters the argument *real x* in the header field of the function *absolute*, it makes an entry into the symbol table. Unlike the entry made during the processing of the **<var>** tag, this entry is prefixed with the function name and an underscore. Hence, in the symbol table, the variable *x* is actually stored as *absolute_x*. Every variable encountered within the function body is stored this way. This gives us two advantages.

1. This allows us to maintain a single symbol table- rather than maintaining a separate symbol table for each function.
2. The user can define variables with the same name as long as they are in different functions.

Translating functions is a bit more complex for Java. As the functions may be called from **main**, the functions themselves must be **static**. Moreover, any function- including main- must throw IOException, as we have used the readLine() function. We now take a look at the output of the same function *absolute* when translated into Java.

```
static private double absolute(double x) throws IOException
{
 if (x >= 0) {
 return x ;

}
else
{
 return (-1 * x) ;
 }


}
```

We note that the function has automatically been defined as **static** and it throws an IOException. ALGOSmart would add the “throws IOException” phrase to every function- even if the function does not have any Input/Output Statements. This is because ALGOSmart- hence our translation process- is a *single pass* algorithm. There is no way for the translator to look ahead and check whether the function body comprises Input/Output statements.

So does the C (or Java) programs generated, by ALGOSmart, after translating the above pseudo code work as expected? We save the pseudo code file as **succ_bis.xml** and then we run the ALGOSmart program. Here's the output:

```
WELCOME TO ALGOSmart!

Enter algorithm file name: succ_bis.xml
Type 1 to convert to C, 0 to convert to Java: 1

Enter output filename (with the extension): succ_bis.c

BUILD SUCCESSFUL!!!
ALGOSmart has finished generating the source code from the XML Specification.

Type 1 if you wish to compile the source code now: 1

Invoking gcc compiler to compile the C code...

Type 1 if you want to run the program now: 1
Enter x0:2.0

Enter x1:7.0


Solution converges to a root:
Number of iterations =10
Root is:5.00293

BYE!
```

Fig 9: An interaction with ALGOSmart

We see that the generated C code computes the root perfectly: 5.00293. A similar interaction with ALGOSmart to translate the pseudo code to Java also yields satisfactory results.

5. Conclusion

In this paper, we have tried to present a translation process where the user presents a pseudo code as an input, and the output is an implementation of the pseudo code in a specific programming language.

The first part of the paper discussed the Algorithm Specification Module. This gives the user a formal framework to specify pseudo code. The pseudo code specification is at a fairly high level and is thus easy to understand. Next, we discussed the Algorithm Translation Module, which takes the pseudo code and translates it into a program. We have already implemented the translator, using Perl, and we call the program ALGOSmart. We took up a representative algorithm- the Successive Bisection Method, wrote the pseudo code, and then analyzed the translation process. The primary feature that we have tried to underline throughout is the insulation that our translator provides to the user from the idiosyncrasies of implementation. Finally, we provided a snapshot of an interaction with ALGOSmart. The snapshot reveals that the translation performed by ALGOSmart on the Successive Bisection Algorithm yields satisfactory results.

The primary areas where the translator may be further developed include providing an interface for handling Files. Presently, the translator ALGOSmart can only read from the keyboard. As the algorithm for operation handling (Fig 6) is a fairly modular one, incorporating this feature should be a fairly straightforward task.

The other area is to extend support for more programming languages. As the only language dependent operation during the entire translation process is limited to only a few steps, this is also straightforward task.

The translation process has been designed keeping scalability in mind, and hence future improvements and expansion of the process is something that we invite.

References

[1] Atkinson K, *Elementary Numerical Analysis*, 2nd edition, John Wiley and Sons.

[2] Chen Shyi-Ming, Lin Chung-Hui, Chen Shi-Jay, *Multiple DNA Sequence Alignment Based on Genetic Algorithms and Divide-and-Conquer Techniques*, International Journal of Applied Science and Engineering 2005. 3, 2: 89-100

[3] Dijkstra E.W, *Algol 60 translation: an Algol 60 translator for the X1*, Mathematisch Centrum, Amsterdam, 1961.

[4] Kruskal J. B, Jr. (1956): On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, **7**, pp. 48-50.

[5] MARST, ALGOL-60 to C translator, http://www.gnu.org/software/marst/

[6] Navarro G, *A Guided Tour to Approximate String Matching*, ACM Computing Surveys, 33(1):31-88, March 2001.

[7] Rajaraman V, *Computer Oriented Numerical Methods*, 3rd edition. Prentice Hall of India- Private Limited.

[8] Randall B, *The Whetstone KDF9 ALGOL Translator*, The English Electric Company Limited, Atomic Power Division, Whetstone, England. http://www.cs.ncl.ac.uk/publications/books/papers/124.pdf

[9] The Language Guide, University of Michigan, Dearborn Computer Science and Information Science Department. http://www.engin.umd.umich.edu/CIS/course.des/cis400/index.html

[10] Wall L, Christiansen T, Orwant J, *Programming Perl*, 3rd edition. O'Reilly pages 10-12.

[11] XML Tutorial- XML Tree at w3schools; http://www.w3schools.com/xml/xml_tree.asp

[12] XML Tutorial- XML DTD at w3schools: http://www.w3schools.com/xml/xml_dtd.asp