A Prototype of FORTRAN-to-Java Converter

Geoffrey Fox, Xiaoming Li*

NPAC at Syracuse University
Syracuse, NY 13244, {gcf,lxm}@npac.syr.edu

Zheng Qiang, Wu Zhigang

Computer Science Department
Harbin Institute of Technology
Harbin, 150001, China, {zq,wzg}@pactpc.hit.edu.cn

July 20, 1997

Abstract

This is a report on a prototype of a FORTRAN 77 to Java converter, *f2j*. Translation issues are identified, approaches are presented, a URL is provided for interested readers to download the package, and some unsolved problems are brought up. *F2j* allows value added to some of the investment on FORTRAN code, in particular, those well established FORTRAN libraries for scientific and engineering computation.

Key words: FORTRAN, Java, Automatic Translation

Acknowledgement: The authors would like to thank the referees of this paper and attendants of the Workshop on Java for Science and Engineering, where the paper was presented, for their insightful comments.

1 Introduction

As Java gets its dominance in Internet programming, it is natural for people to consider how Java may also be used in scientific and engineering computations. As Joe Keller, director of marketing and support/workshop products

*Contacting author, visiting scholar from HIT, China

at Sun, indicated: while the first Java versions were built for portability, the next versions will be built based on performance [10].

Making Java faster is necessary for exploiting full potential of Java language for Internet applications, and many research groups and venders are pursuing various technologies to improve Java's performance. These technologies include, but are not limited to, JIT compilation, optimizing compiler, parallel interpretation, and parallelization of source code, etc. [11]. Thus, Java will be fast, and the faster Java gets, the more scientific and engineering problems can be solved in Java — taking Java's well known advantages, and with acceptable performance. After all, Java now is faster than FORTRAN 20 years ago, and people were doing pretty good science and engineering with FORTRAN then.

While observing that computers are never fast enough to meet the requirements of leading edge science and engineering work, it should be safe to say that many circumstances where FORTRAN is used are not really time critical.

For more than 30 years, FORTRAN is still the fastest language for number crunching. But it seems a tradition that people like to build converters that translate FORTRAN programs to whatever popular languages. Besides the famous *f2c* maintained at Bellcore [7], there was also a FORTRAN to Pascal converter [9] when Pascal was popular. And there are some companies, part of their businesses is to convert FORTRAN programs to others [7]. Any way, there are some good reasons for turning FORTRAN to something more promising. Java's platform neutralness and mobility make it a more attractive language to turn legacy code into.

Thus, we have embarked on the effort of building a FORTRAN to Java converter, with a belief that Java is certainly more useful than Pascal, and will be more widely used than C.

From implementation point of view, the converter is based on `HPFfe` [1] and a previous FORTRAN-to-C conversion work done by one of the authors [8]. `HPFfe` constructs an abstract syntax tree (AST) for input FORTRAN program, a FORTRAN-to-C conversion module then turns this AST to a C counterpart. An enhanced unparsing process finally spells out a Java program from this intermediate AST. Thus, we observe the following process:

$$FORTRAN\ source \xrightarrow{A} FORTRAN\ AST \xrightarrow{B} C\ AST \xrightarrow{C} Java\ source$$

It seems weird to have a C AST involved in the middle. A more natural approach would be defining an intermediate representation (IR) for Java, and turning the AST for a FORTRAN program to the Java IR, followed by a straightforward Java unparsing. The path we took is merely for our convenience, since a FORTRAN-to-C module is already there, working with `HPFfe` in concert, and there is not much difference between C and Java.

From user's point of view, an application in FORTRAN consists of multiple source files, and each file has one or more program units. F2j takes as input a FORTRAN source file, say *module1*.f, and turns it into a semantically equivalent Java source file *module1*.java. Each FORTRAN program unit is translated into a Java class in the file. At the moment, no package information is incorporated in the Java source. Thus, the unnamed default package is assumed.

2 Main issues to be addressed in a FORTRAN-to-Java converter

Although it is true that translating a FORTRAN program to Java program is relatively easier than other way around, some issues have to be dealt carefully for both semantical equivalence of the corresponding programs and efficiency of the resulting Java program. Our experience has shown that some of the issues are non trivial. In fact, we have not reached satisfactory solutions for some of them. In this section, we briefly introduce the main issues that have been addressed in our converter. We describe our translation schemes for each one of them in the next section.

- Naming convention.

  Basically, there are two kinds of names in the resulting Java program. One is *type* name, i.e., *class* name; the other is *variable* name. It is obvious that majority of those names should be derived from names in the input FORTRAN program, based on some convention. Moreover, some additional names have to be created to compensate the discrepancies of the two languages.

- Correspondence between FORTRAN program units (main program, subroutine subprogram, function subprogram, and block data subprogram) and Java classes.

  In fact, this is one of our basic decision, namely making a one-to-one correspondence between FORTRAN program units and Java classes. It may be conceivable to design a many-to-one scheme. But we thought it would make things complicated, though some benefit of it is observed.

- The matching of function/subroutine calls in FORTRAN and method invocations in Java.

  The basic problem is that FORTRAN passes arguments by address, while Java passes arguments by value for primitive data types and by reference for general objects.

- Differences in data types.

  Although Java provides a rich type system and FORTRAN is very primitive in this regard, to effectively represent FORTRAN types in Java needs some design.

  In particular, FORTRAN array presents a major problem. In FORTRAN, array is not really a distinct data type. Instead, it's an 'non encapsulated memory region'. Programmers are allowed to do various 'tricks' within that region, for instance, forming another array from part of the region through subprogram interface. In Java, an array is an object. One can not assign new meaning (give a name) to a part of the object.

- FORTRAN specific statements.

  FORTRAN has some statements, such as `GOTO`, `COMMON`, `EQUIVALENCE`, and various I/O statements, etc., which are not present in modern languages. We need to find proper Java correspondence for them.

In what follows, our discussion will be focused on translation schemes for the above issues. Coding details will not be discussed, since it is closely related to the data structure of the AST.

3 The translation approaches

From top level, a `FORTRAN` library is converted into a Java package, a `FORTRAN` file, *filename.f*, is converted into a `Java` file, *filename.java*, and each `FORTRAN` program unit is turned to a Java class.

Besides, the following issues are addressed in the converter.

3.1 Naming conventions

Following rules are made for the formation of names in resulting Java programs.

- Since a Java class is generated for each FORTRAN function or subroutine, the name of the class is the name of the original function/subroutine name with '_c' attached to it. 'c' stands for 'class'.

  For example: a function in a FORTRAN source code named `func1` will be converted to a Java class with the name `func1_c`.

  The `FORTRAN` main program is also converted into a class, but '_mc' is attached to its name to form the Java class name.

- A class variable is generated for each scalar dummy argument in order to solve the problem of argument passing. The name of the class variable is the name of the original argument with '_cv' (class variable) appended to it. For example: class variable `para1_cv` is generated for the argument `para1`.

  This is just what we have implemented, not a perfect solution. There are some non trivial subtlety here. We'll discuss more about it in section 3.3.

- 'j' is inserted before each statement label in FORTRAN source code to form a Java statement lable.

  For example: label 10 will be converted to j10 in Java program.

- Other names in the Java program are identical to those in FORTRAN.

3.2 How to produce a class

A primary difference between object oriented language and procedural language is the former introduces the powerful concept of *class*. Many other differences are derived from it. Class is a basic concept in Java [3], but it does not exist in FORTRAN 77 [6]. Although the direction of this difference does not present major difficulty for our job, since class is a more general concept than procedure, some details have to be taken care.

In order to successfully convert a FORTRAN source to Java source, classes has to be produced. But how to do it? At least two approaches can be considered.

- A class is generated for a whole FORTRAN source file with methods in the class corresponding to functions/subroutines in the FORTRAN file;
- A class is generated for each function/subroutine. It contains a "`public static`" method which is semantically equivalent to the original function/subroutine.

Less `.class` file will be generated and higher performance will be achieved (due to less dynamic run-time loadings) if the first method is used, but some difficulties will be brought into function/subroutine invocation and parameter passing, which we do not have a clear idea yet. So the second method has been adopted. Besides, the name rules specified above are observed with this method.

As an example, the following FORTRAN subroutine

```
subroutine signMeUp(price, price1)
integer price, price1
price = price1 + 1
return
end
```

will be converted into:

```
class signMeUp_c {
   static int price_cv;
   static int price1_cv;
   public static signMeUp(int price,int price1) {
      price=price1+1 ;
      price_cv = price;
      price1_cv = price1;
      return;
   }
}
```

As we see, two extra class variables are produced. Their use is to solve argument passing problem as described below.

3.3 Subprogram invocation mechanism

Different mechanisms for argument passing is a major issue for the translation. In FORTRAN, when a function/subroutine is called, the addresses of the actual arguments (variables) are passed to it [6]. In Java, the values of the arguments are passed for primitive data types and the references (a kind of value) are passed for objects [3]. This means, in Java, the values of the actual argument variables will not get changed upon returning from methods, while FORTRAN often expects a change.

There is a similar problem when converting FORTRAN to C. There, pointers are used to solve it [7, 8]. But in Java, there is no pointer.

Two approaches were considered.

1. The first method is based on the following facts:

   Java passes non primitive data (object) to methods by reference — a kind of address. Thus, if the reference is not modified in the method, i.e., not appearing at left hand side of an assignment statement, the modification to the member of the object is observed by the caller [3].

   So, we might declare a class for each FORTRAN data type, for example:

   ```
   class INTEGER {
      public int value;
   };
   ```

   and put all this kind of classes into one package named `data_type`. This package should be imported in every produced `.java` file. Then, every variable declaration statement should be converted to the class declaration statement. When a function/subroutine is invoked, the corresponding object should be passed into the function/subroutine. And in the invoked function, memory is not reallocated to the object [3]. The member of the object is modified if the argument in the source FORTRAN file is modified.

   For instance, the following FORTRAN program:

   ```
   program main
      integer a, b
      call xx(a,b)
   end

   subroutine xx(d,e)
      integer d, e
   ```

```
    d = 3
    e = 4
    return
  end
```

yields the following Java program:

```
class main_mc {
   public static void main(String args[]) {
      INTEGER a = new INTEGER();
      INTEGER b = new INTEGER();
      xx_c.xx(a,b);
   }
}

class xx_c  {
   public static void xx(INTEGER d, INTEGER e) {
      d.value = 3;
      e.value = 4;
      return;
   }
}
```

This approach is easy to implement, but not efficient, since objects are artificially created and accessing object is much slower (about 3 times) than primitive type access in Java. We did not use this method in our implementation.

2. The second approach.

   As mentioned above, a class is generated for each function/subroutine. A method semantically equivalent to the function/subroutine is contained in this class. The second approach introduces some class variables into the class, besides the method. Each class variable, which is generated according to the arguments, serves as an intermedium between actual argument and dummy argument: before the function/subroutine returns, the class variable is assigned the value of argument if it is modified in the function/subroutine; after the invocation statement in the caller, the actual argument is assigned the value of the class variable. The names of the class variables are the names of the arguments with '_cv' appended to it.

For the same FORTRAN program above, the following Java program is produced under this scheme.

```
class main_mc {
   public static void main(String args[]) {
      int a=0, b=0;
      xx_c.xx(a,b);
      a = xx_c.d_cv; // produced by converter, modify actual arg
      b = xx_c.e_cv;
   }
}

class xx_c  {
   static int d_cv;
   static int e_cv;
   public static void xx(int d,int e) {
      d = 3;
      e = 4;
      d_cv = d;
      e_cv = e;
      return;
   }
}
```

This scheme is currently implemented in our converter. Notice how the `CALL` statement in FORTRAN is converted to corresponding method invocation in Java.

While being more efficient, this method also suffers from a few problems (we thank one of the referees who pointed out some of them). The first problem is that it does not support separate conversion of `FORTRAN` program unit, namely, names of dummy arguments of a callee must be known when converting a caller. The second problem is incapability of handling dummy arguments' aliasing, namely things like `CALL FOO(a,a)`. In this case, two dummy arguments refer to the same memory location, the order of updating the two dummies in callee determines the value that caller will see after the subroutine returns. But the order of assigning class variables to actual arguments is normally fixed. The third problem is thread safety. Unprotected static class variables may be accessed concurrently in an unpredictable way, when the class is used by multiple threads.

FORTRAN	Java
Integer	int
Real	float
Double precision	double
Complex	class Complex
Logical	boolean
Character	String

Table 1: Mapping between data types

3.4 About data types

Table 1 gives a mapping between `FORTRAN` and Java data types.

`FORTRAN` *complex* and *character* types need some special treatment.

- Complex data type.

  The issue is that Java does not have *complex* type and it does not support operator overloading. Thus, we have defined a class named `Complex`, which includes two data fields for real and imaginary parts of a complex quantity and methods corresponding to primitive arithmetic operations (+, −, *, /). Moreover, a simple *copy* method is included to mimic assignment between two complex variables. (the standard clone() method seems unnecessarily complicated to use for our purpose.)

  For the following example,

```
program complx
complex com1,com2
com1 = (1.2,2.3)+(2.3,2.2)*(2.3,2.5)
com2 = (1.0,1.0)/(1.0,1.0)*(1.0,1.0)-(4.3,3.4)
com2 = com1
end
```

  Corresponding Java program looks like,

```
class complx_mc {
   public static void main(String args[]) {
      Complex com1,com2;
```

```
        com1 = ((new Complex((float)1.2,(float)2.3)))
             .add((new Complex((float)2.3,(float)2.2))
             .mult((float)2.3,(float)2.5));
        com2 = ((new Complex((float)1.0,(float)1.0))
             .div((float)1.0,(float)1.0).mult((float)1.0,(float)1.0))
             .minus((new Complex((float)4.3,(float)3.4))) ;
        com2 = com1.copy();
    }
}
```

- Character strings

  FORTRAN `character` data are fixed length strings of characters. Java `String` has variable length. There are two possible ways to map FORTRAN character data to Java elements, either to `char` arrays, or to `Strings`. We decided on the latter. Thus, the following,

  ```
  character * 10 s1, s2
  s1 = '1234567890'
  s2 = s1(2:4)//'abc'
  s1(2:4) = 'xyz'
  ```

  is translated into

  ```
  String s1, s2;
  s1 = "1234567890";
  s2 = s1.substring(1,4) + "abc";
  s1 = s1.substring(0,1) + "xyz" + s1.substring(4,s1.length());
  ```

  Note that we have taken care of the difference between substring designations in FORTRAN and Java. The fact that `String` object is read-only does not hurt here, since a new object is created and old one is to be garbage collected automatically.

- Arrays

  There are some simple issues such as array declaration and element accessing within the program unit where the array is declared. They can be treated readily. For instance, the following program

  ```
  program foo
  integer  a(1:10)
  ```

```
   integer b(1:10,-10:10)
   integer c(1:10,-10:10,-10:0)
   integer  i,j,k

   do 10 i=1,10
      a(i)=10 - i
      do 10 j=10,-10,-1
         b(i,j) = i + j
         do 10 k = -10,0,2
10          c(i,j,k) = i + j - k * i / j
   end
```

is translated by our converter into:

```
   class foo_mc {
      public static void main(String args[]) {
         int a[] = new int [10-1+1] ;
         int b[][] = new int [10-1+1][10-(-10)+1] ;
         int c[][][] = new int [10-1+1][10-(-10)+1][0-(-10)+1] ;
         int i,j,k ;
         for (i=1; i<=10; i=i+1) {
            a[i-1] = 10-i ;
            for (j=10; j>=-10; j=j-1) {
               b[i-1][j-(-10)] = i+j ;
               for (k=-10; k<=0; k=k+2)
   j10:             c[i-1][j-(-10)][k-(-10)] = i+j-k*i/j ;
            }
         }
      }
   }
```

However, the major problem occurs when passing arrays to subprograms. This is because Java array is 'semantically' different from FORTRAN array (and also different from C array). A FORTRAN array is a collection of *consecutive* memory locations, organized according to dimensioning information. And a Java array is not required to keep its elements together. In fact, we should expect the elements of a multiple dimensional Java array scattered in the memory, (see section 15.9.1 of reference [3]). As a result there is no concept of *storage majority* for Java arrays.

This makes some well established practice in FORTRAN programming, which takes the advantage of consecutiveness of array elements, hard to have an effective Java counterpart. The following is an example (provided by one of attendants of the workshop).

```
REAL X(10)
...
CALL F(X,10)
CALL F(X(3),8)
...
END

SUBROUTINE F(A,N)
REAL A(N)
...
END
```

Things will be even more interesting, if multiple dimension arrays are involved as arguments. We have not come to a satisfactory scheme yet. An attractive solution is to linearize all arrays to single dimension and pass array name together with a starting index to method.

3.5 FORTRAN specific statements

We describe the scheme used in f2j for translating `COMMON`, `EQUIVALENCE`, and labeled `DO` statements.

COMMON statement `COMMON` statements are widely used in FORTRAN libraries as a means to implement global data, and allow storage sharing among different program units. In Java, public static variables can be accessed and shared from all other classes. So we convert a `COMMON` block into a class with variables in the `COMMON` block being translated into `public static` variables in the class.

In this case, some additional naming rules are needed:

- Blank common block is converted into class named `NonameCommon`;
- Named common block retains its name as the class name;
- The members of the class derived from common block inherit the original variable names from the first occurrence of the common block declaration.

As an example, the following FORTRAN program,

```
program comm
integer a,b,fff,i1,i2
real d,f,f10
common a,b,d
common /c0/fff,f
a =0
b =10
d = 5.0
fff = 987.0
f = 3.456
end

integer function myfunc()
integer b,d,i
real c,f1,f2
common b,d,c
common /c0/i,f1
b = 10
d = 5
c = 0.1
i = 3
f1 =19.844
myfunc = d + b
end
```

is translated into

```
class NonameCommon {
   public static int a;
   public static int b;
   public static float  d;
}

class c0_c {
   public static int fff;
   public static float  f;
}

class comm_mc {
   public static void main(String args[]) {
      myfunc_c myfunc_o ;
      int ReplaceMentVar0,i1,i2 ;
```

```
        float ReplaceMentVar1,f10 ;

        NonameCommon.a = 0 ;
        NonameCommon.b = 10 ;
        NonameCommon.d = (float)5.0 ;
        c0_c.fff = (float)987.0 ;
        c0_c.f = (float)3.456 ;
    }
}

class myfunc_c {
    public static int myfunc() {
        int  ReplaceMentVar2 ;
        float  ReplaceMentVar3,f2 ;

        NonameCommon.a = 10 ;
        NonameCommon.b = 5 ;
        NonameCommon.d = (float)0.1 ;
        c0_c.fff = 3 ;
        c0_c.f = (float)19.844 ;
        return(NonameCommon.b+NonameCommon.a);
    }
}
```

This scheme solves global variable problem, and partly solves storage sharing problem. It fails when two corresponding common variables have different type, such as in

```
PROGRAM MAIN
INTEGER A, B, C
COMMON /C1/A,B,C
...
END

SUBROUTINE FOO()
REAL X,Y,Z
COMMON /C1/X,Y,Z
...
END
```

A similar drawback is also associated with our current treatment for `EQUIVALENCE` statement.

EQUIVALENCE statement In our converted program, `EQUIVALENCE` statement is treated in a simple minded fashion, namely, when the variables are accessed, we replace them with the variable's name which first appears in the EQUIVALENCE statement. Thus, the following program,

```
program main
integer a,b,c
equivalence (a,b)

a = 10
b = 9
c = 8
end
```

is translated into

```
class main_mc {
   public static  void main(String args[]) {
      int a,b,c; //b is of no use, but is kept
      a = 10 ;
      a = 9 ;
      c = 8;
   }
}
```

Labeled DO statement It is not difficult to deal with labeled DO statements, since the labels for nested DO loops must be properly nested, which have a natural correspondence to nested FOR loops in Java. We simply keep those labels (of course change them to Java labels) and add proper '{' and '}'.

The following program,

```
      program bbb
      integer a(10)
      integer b(10,10)
      integer c(10,10,10)
      integer i,j,k
      do 20 i=1,10
         do 10 j=10,1,-1
            b(i,j) = i + j
            do 10 k = 1,10,2
10            c(i,j,k) = i + j - k * i / j
20            a(i)=10 - i
      end
```

becomes,

```
   class bbb_mc {
      public static  void main(String args[]) {
         int a[] = new int [10] ;
         int b[][] = new int [10][10] ;
         int c[][][] = new int [10][10][10] ;
         int i,j,k ;
         for (i=1;i<=10;i=i+1) {
            for (j=10;j>=1;j=j-1) {
               b[i-1][j-1]=i+j ;
               for (k=1;k<=10;k=k+2)
j10:              c[i-1][j-1][k-1]=i+j-k*i/j ;
            }
j20:        a[i-1]=10-i ;
         }
      }
   }
```

3.6 I/O and FORMAT statement

We have implemented translation of three kinds of I/O statements in their primitive forms: `WRITE`, `PRINT` and `READ`. FORTRAN provides sophisticated formatting facility for I/O through *edit descriptors* [6, page 13-5], while Java does not [4, page 189]. This discrepancy makes it difficult to faithfully translate FORTRAN I/O statements to Java. Thus, in our first attempt, formatting information is largely ignored. Nevertheless, some "important" formatting information, such as data items interleaved with pre-specified character strings, is properly translated. Thus, for the FORTRAN program:

```
      program io
      integer a,b,c,d,e
100   format (i3,'Happy Day!')
200   format (i5,'Get',i8,'Hello')
      a=9
      b=6
      c=5
      write(*,200) a,b,c
      print 200, a,b,c
      write(*,200) a,b,b,c
      print 200, a,b,c,d
      write(*,100) a,b,c
      print 100, a,b,c
```

```
      write(*,*) 'A=',a,'B=',b,'C'
      print *, 'A=',a,'B=',b,'C'
      write(*,'(i5,i9)') a,b
      print '(i5,i9)', a,b
      end
```

f2j converts it into:

```
   class io_mc {
      public static  void main(String args[]) {
         int a,b,c,d,e ;
         a=9 ;
         b=6 ;
         c=5 ;
         System.out.println(a+" "+"Get"+" "+b+" "+"Hello"+"\n"+c+" "+"Get");
         System.out.println(a+" "+"Get"+" "+b+" "+"Hello"+"\n"+c+" "+"Get");
         System.out.println(a+" "+"Get"+" "+b+" "+"Hello"+"\n"
                           +b+" "+"Get"+" "+c+" "+"Hello");
         System.out.println(a+" "+"Get"+" "+b+" "+"Hello"+"\n"
                           +b+" "+"Get"+" "+c+" "+"Hello");
         System.out.println(a+" "+"Happy Day!"+"\n"+b+" "+"Happy Day!"+"\n"
                           +c+" "+"Happy Day!");
         System.out.println(a+" "+"Happy Day!"+"\n"+b+" "+"Happy Day!"+"\n"
                           +c+" "+"Happy Day!");
         System.out.println("A="+" "+a+" "+"B="+" "+b+" "+"C");
         System.out.println("A="+" "+a+" "+"B="+" "+b+" "+"C");
         System.out.println(a+" "+b);
         System.out.println(a+" "+b);
      }
  }
```

Notice that we have inserted a blank space between two consecutive data items, and we have translated the effect of cyclic use of formatting rules.

4 Current limitations and considerations for further improvement

While a prototype package can be download from http://www.npac.syr.edu/projects/pcrc/f2j.html for interested readers to play with, some more work is needed for the *f2j* to be truly usable. Besides the following items identified for further work, we plan to incorporate the f2j into a web server, so that a user does not have to download the package. Instead, he submits his FORTRAN program to our f2j server, and it will email him back a Java program.

- Array processing. As mentioned previously, current *f2j* can only handle arrays within one program unit. We need to implement a scheme that is able to translate arrays as dummy arguments to subprograms.
- `GOTO` statement was left untouched in current *f2j*. It seems feasible to convert `GOTO` effectively, using Java *break*, *continue*, and *try-catch-finally* construct.
- A Java package which is functionally equivalent to FORTRAN intrinsic functions should be constructed, which will surely have a lot to do with class `java.lang.Math`.
- Although basic I/O statements have been translated, those associated with file operations need to be covered, as well as a reasonable coverage of `FORMAT` statement.
- `BLOCK DATA` subprogram is not supported at the moment.
- About argument passing between program units, the current implementation has three problems as discussed previously. They are to be solved.
- The current treatment for `COMMON` and `EQUIVALENCE` statements are not general enough. Sequence and storage association issue involved with these two statements is not addressed.

References

[1] Xiaoming Li, et al, "HPFfe: a Front-end for HPF," NPAC Technical Report, SCCS-771, May 1996.

[2] Li Xinying, "HPF front end technical report," PACT technical report, May 1996.

[3] James Gosling, Bill Joy, and Guy Steele, The Java Language Specification. Addison-Wesley, Reading, Massachusetts, 1996.

[4] Ken Arnold and James Gosling, The Java Programming Language. Addison-Wesley, Readings, Massachusetts, 1996.

[5] Tan Haoqiang, FORTRAN 77 structural programming. Tsinghua Publishing House.

[6] ANSI X3.9-1978, ISO 1539-1980(E), American National Standard Programming Language FORTRAN. American National Standards Institute, April 3, 1978.

[7] S. I. Feldman, David M. Gay, Mark W. Maimone, N.L. Scryer, "A Fotran to C Converter," Computing Science Technical Report No. 149, Bellcore, Morristown, NJ, May 16, 1990, last updated March 22, 1995.

[8] Qiang Zheng, "A FORTRAN to C translator based on Sigma system," PACT technical report (in Chinese), July, 1994.

[9] R. A. Freak, "A FORTRAN to Pascal Translator," Software — Practice and experience, Vol. 11, 1981, 717-732.

[10] Krista Ostertag, "Java: Beyond the Hype," VARBusiness, March 1, 1997, 63-64.

[11] Geoffrey C. Fox (ed.), Concurrency: Practice and experience, Vol. 9, No. 6, June 1997, a special issue on Java for computational science and engineering — simulation and modeling.