University of Nevada

Reno

# Large Scale Software Transitions:
# A Case Study of the First Half of MFIRE

A professional paper submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science

by

Lingjiang Cheng

Dr. Frederick C. Harris, Jr., advisor

May 2000

# Abstract

This paper presents a case study of large scale software transitions. This is the first part our virtual reality of mine ventilation systems project. This project is based on an existing software package called MFIRE. “MFIRE is a computer simulation program that performs normal ventilation network planning calculations and dynamic transient state simulation of ventilation networks under a variety of conditions. The program is useful for the analysis of ventilation networks under thermal and mechanical influence. MFIRE simulates a mine's ventilation system and its response to altered ventilation parameters: external influences such as temperatures, and internal influences such as fires. “[16]

MFIRE was written in FORTRAN. Modern virtual reality software and utilities are typically written in C++, an object orientated programming language. Since communication between C++ and FORTRAN is difficult and the ability to modify the code is vital, we needed to convert the existing FORTRAN code to C++ with the exact same functionality.

This paper also discusses the transition methodology and the debugging of the code.

# Acknowledgements

I would like to express deepest and sincerest gratitude to my advisor, Dr. Fred Harris, for all his help, guidance and encourage throughout my academic career in Computer Science at University of Nevada, Reno. His excellent class of CS 308 data structures in C++ sparked my interest in Computer Sciences and encouraged my final decision to pursue a continuing education in Computer Science.

I also wish to thank Dr. George Bebis and Dr. Bruce Johnson for being on my committee and their valuable time.

Finally, a special thank to my wife, Jiali, for her encouragement and support.

# Contents

## List of Figures

# List of Tables

# 1. Introduction

MFIRE is a computer simulation program that performs normal ventilation network planning calculations, and dynamic transient state simulation of ventilation networks under a variety of conditions. The program is useful for the analysis of ventilation networks under thermal and mechanical influence. MFIRE simulates a mine's ventilation system and its response to altered ventilation parameters, external influences such as temperatures, and internal influences such as fires. Extensive output enables detailed quantitative analysis of the effects of the proposed alteration to the ventilation system.

Network simulation using digital computers has become widespread throughout the mining industry. However, as the sophistication of the simulator increases (MFIRE, for example), the complexity of input data requirements and interpretations of results requires more skill and knowledge from the users. Perhaps the most difficult part of using MFIRE is to construct the data set describing the mine's physical layout and its ventilation properties. The initial attempts to get the data set running often reveal unknown or ignored aspects of the mine's ventilation[16].

MFIRE was written in Fortran 77. Our purpose is to use the output of MFIRE and display the mine and its ventilation three-dimensionally on a computer screen. This paper is the first part of the whole project. In Section 2 we present a background on mine ventilation, software to do it, and the development of MFIRE. Section 3 presents some of the differences between FORTRAN and C++ that we had to deal with. Section 4 covers our code transition and debugging of our code is presented in Section 5. Conclusion and Future work are discussed in Section 6.

# 2. The Background of MFIRE

## 2.1 What is MFIRE

Mine ventilation control and mine fire detection and fighting are inseparable. Mine fires produce gases and heat, which the ventilation systems transport through the mines. These gases can be poisonous and/or explosive[5,6,7]. The heat can cause ventilation disturbances, which take the gases along unexpected routes or affect the formation of explosive methane mixtures.

The calculation of the airflow distribution in mine ventilation systems as a result of fans, thermal forces, and flow resistances is a formidable mathematical problem[3,4]. It comprises the solution of twice as many equations as there are airways; half of these equations are quadratic equations[4]. This sort of problem led to the design of special analog computers in the 1950's and 60's and, from the early 1960's on, to the increasing use of electronic digital computers[11]. With the rapidly increasing availability and capacity of digital computers, airflow rate and pressure loss distribution calculations, commonly called ventilation network calculations, have become routine, and a great number of computer programs exist for this purpose[13]. Practically all the programs are capable of performing the required calculations, although differences exist in how the square equations are linearized, the mass conservation law is introduced and observed, the fan characteristics are simulated, and the thermal drafts are considered. All of the early programs were based on steady-state conditions.

Of greatest concern in the past were the fire-generated ventilation disturbances[8,9,10,11]. Ventilation engineers developed a large number of methods, by manual calculation, to detect potentially unstable airways with airflow reversals in case of a fire[5]. When the analog and digital computers became available for ventilation planning, they were almost immediately applied to this problem. The expected fire-generated ventilating pressures were manually

inserted into the network simulations, with their values usually obtained from experience or from rough calculations. The mutual influence of fire intensities and ventilation conditions were not taken into account. If gas concentrations were calculated at all, they were only calculated for the cases where no recirculation existed[12,13]. All calculations were, as in conventional network calculations, based on steady-state conditions or based on the assumption that no changes with time occur.

The U.S. Bureau of Mines and Michigan Technological University first solved this problem with steady-state analysis, and the resulting program became known as the MTU/BOM code. Because the mine fire process is dynamic in nature, work on the transient-state modeling problem continued. The resulting program, MFIRE[16], accommodates dynamic state modeling of the fire[14].

MFIRE version 2.20[16], which was finished in 1995, includes calculations based on mass flow rates, natural ventilation, `spline` or least squares fan curve fitting and boundary fixing, air reversal and `recirc`ulation calculations. Condensation and evaporation in the mass flow and heat exchange calculations was removed from MFIRE 2.20 but will hopefully be incorporated in a later version. The program is useful for the analysis of ventilation networks under the influence of natural ventilation, fans, fires, or any combination of these. MFIRE simulates a mine's ventilation system and its response to altered ventilation parameters such as the development of new mine workings or changes in ventilation control structures, external influences, and internal influences. Extensive output enables detailed, quantitative analysis that the alteration will cause in the ventilation system.

## 2.2 The History Leading up to MFIRE Development

Ventilation network calculations have been performed for several centuries[15]. Due to mathematical difficulties caused by diagonal airways, the

preferred method of practitioners was a trial and error approach in which junction and mesh equations were made compatible. Since the beginning of this century, airflow and pressure loss distributions in a large number of more frequently occurring subnetworks were analytically and graphically determined. These were the subjects of numerous publications and were of considerable help to practitioners[3,4,5,6,8,9].

Trial and error methods, which can be surprisingly efficient in one case, can become frustratingly inefficient in another. A large number of methods of successive approximation were consequently developed. Atkinson's solution in 1854 for a single diagonal airway and Cross's method because of its general applicability and simplicity became the widest known examples[3,4]. Some of these methods were based on the linearization of the quadratic resistance equation and used in electric analog computers. Practically all of the methods were tested for their utility with digital computers when these became available.

The forerunners of analog computers were fluid flow models. They were used in several countries but never found wide application. The similarity between node and mesh equations in ventilation networks with Kirchoff's laws of electrical networks made the electrical models persuasive. The first patent for an electric analog computer for water and gas networks was awarded in 1941 in Germany. This computer used filament bulb resistance to model the second power resistance function in the network calculations[8,9].

The idea was taken up or independently discovered in several countries, but the limited working range of commercially-available filament bulbs made them inflexible and the number installed remained small. Development of a specially-constructed low voltage lamp in the United States overcame this obstacle and in 1954 a "network analyzer" was installed at the U.S. Bureau of Mines, after six earlier installations had shown their usefulness with waterworks[14].

In 1951-52, the University of Nottingham (UK) pioneered the idea to combine an electrical network simulation of the nodes and mesh equations of

ventilation networks with a manual approximation method for the resistance equation. This led to the design of the commercially available "National Coal Board Network Analyzer", which found a wide distribution[14]. Home-built models and modifications, using different approximation methods for the resistance equation, were used in almost all mining countries. At some places ancillaries for adjustment steps without calculations were introduced.

An electromechanical analog computer, in which the approximation of the resistance equation was automatically performed, was first developed in 1950 at a German coal mine. It became commercially available in 1952. Thirteen of these computers were installed in German coal mines and a larger number were installed abroad and for gas and water companies. In 1959, electronic function generators for the simulation of the resistance equation were introduced in Japan; in 1960 the German manufacturer adopted this principle, also. In 1964, a British model became commercially available and may be the only one still on the market. In 1962, the French coal mines built an electronic model which has been used for several decades[11].

Fully automatic analog computers for ventilation network calculations are excellent planning tools. Their handicap is that they are single purpose machines. All-purpose digital computers became commercially available in the late 1950's and predictably replaced the majority of the analog computers.

The literature reports that the first network calculations with digital computers were performed for waterworks in the United States in 1957[10]. The first digital ventilation network calculations were reported in Belgium in 1958 and in Germany in 1959. Following the lead of gas and water companies, efforts to replace the expensive analog computers with digital computers began in Germany in 1958. The replacement progressed quite rapidly since many of the analog computer users were cooperating with this effort. The literature reports that the same coal company that had pioneered the use of electromechanical analog computers performed almost all of their network calculations on digital computers

by the end of 1959. By the end of 1969 the majority of analog computer users, representing 80 % of the German coal production, had switched to digital computers[10,11].

Since the first attempts with the Cross method of balancing pressure losses gave poor convergence most other known approximation methods were initially attempted. These were dropped when it was found that you could overcome the convergence problem by assembling meshes in such a way that airways with high resistance factors (or even better, with high products of airflow rates and resistance factors) occurred in as few meshes as possible. The mesh assembly was done in a systematic way by arranging a tree in a sequence from tree tip to root, which allows the computer to elect correct movements when assembling the meshes[4,5].

From 1961 on, it became customary to include natural ventilation obtained from information on temperatures and elevations in every mesh. Fixed quantity airways had always been a feature of analog computers and were included in the earliest applications of digital computers. Fan characteristics were treated in different ways as storage allowed. A FORTRAN version of a type of standard program became a part of the IBM program library in 1966; in 1967 it was adopted by the British National Coal Board for ventilation planning purposes at its divisional computer centers. It has been used for instructional purposes at Michigan Technological University (MTU) since 1967. Although many enhancements and attempts at improvements were made, it is basically still in its original form and is the core of the MFIRE program[2,3].

As the availability of digital computers increased, the number of users doing creative work in ventilation planning increased tremendously. Due to personal, societal, or company restraints, much work went unreported. However, by analyzing available published literature, it is clear that continental European ventilation engineers cooperated closely. An advanced program capable of performing high-speed calculations for large networks was in use in France in

1961. A storage saving program based on the Cross method of flow rate balancing was introduced in 1967. In Japan network calculations with digital computers started in 1961. Convergence-improving mesh assemblies were reported in 1969. In Russia, first attempts with digital computers were made in 1963. In 1965 and 1967, reports on different approximation methods were published. In Great Britain, the first network calculations with the meshes assembled manually were reported in 1964. A program with automatic mesh assembly was described in 1965. In the United States, the first program to prove the usefulness of digital computers was described in 1963; an improved version allowing the inclusion of fan characteristics was reported in 1964. Both programs still required the manual assembly of meshes[12,13].

In 1967, a much more sophisticated program with mesh assembly and assignment of initial airflow rates performed by the computer was reported and in 1970 a new version of the program which accommodated fixed quantity airways was described[9].

Over the past two decades efforts focused on: (1) replacing the Cross method with more efficient approximation methods, (2) combining network calculation for optimization purposes with operations research approaches, (3) making the programs more user friendly in particular by using interactive graphics, (4) combining network calculations with temperature and concentration calculations, and (5) extending network calculations to transient state conditions[2].

The first objective has been a continual goal since the first days of digital computer use. So far, all results seem to confirm that the Cross method for networks of ordinary size and complexity is as good or better than other methods. The second objective is a very valid one since network calculations are only a means to an end. The third objective is probably the most important one[8].

The non-steady-state behavior of ventilation systems has attracted research in connection with control problems since the 1950's. Studies concentrated upon

the effects of explosions, gas outbursts, and other mechanical disturbances[10]. A publication describing the use of digital computers for this purpose originated in Poland in 1972[9] and another one using analog computers was published in Yugoslavia in 1984[13].

Efforts to combine ventilation network calculations with the precalculation of temperatures and humidity started in Japan in 1969. An early program that included temperatures, humidity, methane and dust concentrations, plus a transient state methane simulator originated at the University of Pittsburgh in 1972. In 1975, at the First International Mine Ventilation Congress, reports on four programs from the United States and Great Britain for combined network, temperatures and humidity calculations were given. At the Third Congress in 1984, a program for temperature, humidity, and radon concentrations was introduced from Australia[13].

Litigation connected with the Sunshine mine fire in the mid-1970's showed that existing programs could only partially simulate the interaction of mine fires and ventilation systems. Although manual non-steady-state temperature precalculations had become a common feature and steady-state fume concentrations were easy to add as long as no recirculation occurred, manual insertions of thermal draft, and throttling effects proved to be cumbersome and the handling of recirculation to be impossible[2].

This led to the development of a new program at MTU in 1975 and 1976. The goals of this program were to determine the equilibrium between fires and ventilation systems in steady- state conditions at any given time. The crucial heat exchange between rock and air were calculated under non-steady-state conditions. The program was based on mass flow rates and considered natural ventilation in all meshes and throttling effects in all airways. Airflow reversal and fume recirculation were also calculated. This program, sometimes referred to as the MTU/BOM code, was the primary building block of MFIRE[16].

# 3. Differences between FORTRAN and C++ Code

## 3.1 Data Types

In FORTRAN, there are three basic data types, `INTEGER`, `REAL`, and `CHARACTER`. They are corresponding to the `int`, `float`, and `char` data types in C++. The `INTEGER` and `REAL` variables need not to be defined and can be used directly. Any variable begins with letter `I-N` is an integer variable. Others are `REAL` variables. In our code transition, we define all the variables begin with `I-N` as `int`. Others are defined as `double`.

FORTRAN and C++ both support arrays. The subscripts of FORTRAN arrays begin at 1 while C++ arrays begin at 0. For simplification of our transition and avoiding artificial errors, we define arrays in C++ one element larger than the arrays in FORTRAN and do not use the element subscribed with 0. Thus the subscripts are the same in FORTRAN and C++.

## 3.2 Input and Output

FORTRAN uses `READ` statements to input data from console and data file; and uses `WRITE` statements to output information to console and output file. FORTRAN reserves unit 5 for console input and unit 6 for console output. Data files and output files are opened with `OPEN` statements and are associated with a unit number at the time of open. The opened files are good for input and output at the same time. But you should be careful if you consider doing both input and output operations against a same file.

FORTRAN inputs and outputs are formatted. Incorrect format of input data will result in errors. List directed inputs and outputs are acceptable and should be

assigned at time of code design. List directed console input and output converts data to ASCII or from ASCII automatically. File input and output in list directed form are treated in binary code to save disc space and accelerate the computation time. The binary form of data can be read only by FORTRAN code. `OPEN`, `REWIND`, `BACKSPACE`, `ENDFILE`, and `CLOSE` statements are available for file operations.

C++ generally uses `iostream` instances to handle input and output operations. Two instances `cin` and `cout` are reserved for console input and output. File input and output are handled by user-defined instances of `ifstream` and `ofstream` classes. C++ input and output are unformatted. Specific formats are manipulated with `iostreanm` manipulators, such as `setw`(n), `setprecisions`(n). By default, C++ `iostreams` communicate with a file in ASCII mode.

Since the data formats are different, it is difficult to communicate between C++ code and FORTRAN code thought the use of intermediate data file. Direct function calls between C++ code and FORTRAN code are also difficult and troublesome. And since we wanted to modify the code, it was necessary to convert the existing FORTRAN code to C++.

### 3.3 `SUBROUTINES` and `functions`

FORTRAN `SUBROUTINES` and `FUNCTIONS` are corresponding to the functions in C++. FORTRAN `SUBROUTINES` are functions that do not return values and are corresponding `void` functions in C++. By default, the parameters of FORTRAN `SUBROUTINES` and `FUNCTIONS` are passed by reference. If a `SUBROUTINE` or `FUNCTION` is called with (a) value parameter(s), the parameter(s) is/are temporarily passed by value. By default, every variable that

appears in a FORTRAN `SUBROUTINE` or `FUNCTION` is a local variable, and **every local variable is static**. This means that the value of a local variable is retained between `SUBROUTINE` and/or `FUNCTION` calls. There are no "*global*" variables in FORTRAN. FORTRAN uses `COMMON` statements to allow variables in different `FUNCTIONS` and `SUBROUTINES` to share the same memory space. Since `COMMON` variables in different `SUBROUTINES` and `FUNCTIONS` occupy the same memory space, the variables can be considered as global variables.

In our code transition, we initially allowed all the parameters to be passed by reference. Since in C++, a constant parameter can not be passed by reference, we need to fix parameters that have to be passed by value. The situation was picked out at compilation time. All variables declared in the `COMMON` statements are declared as global variables. Others are declared as local variables. It is not necessary to declare all the local variables to be `static`. We had to fix a few of the variable at debugging time. In our code conversion, we found only a few variables that needed to be declared as `static` variables.

### 3.4 Arithmetic Operators

In FORTRAN and C++, the arithmetic operators (+, -, *, /) are the same and have the same precedence order. FORTRAN has an exponentiation operator (**). C++ has no corresponding operator. We chose to use a standard function `pow`(*double*, *double*) instead.

### 3.5 Relational Operators

The relational operators are different between FORTRAN and C++. There

is a simple correspondence, or mapping, from one language to the other and the operators have the same precedence order. Table 1 gives the operators:

### 3.6 Flow of Control

**Table 1:** The relational operators in FORTRAN and C++

| FORTRAN | C++ |
|---|---|
| .EQ. | == |
| .NE. | != |
| .LT. | < |
| .LE. | <= |
| .GT. | > |
| .GE. | >= |
| .NOT. | ! |
| .OR. | \|\| |
| .AND. | && |

The flow control statements of FORTRAN and C++ are quite different. The basic flow control statements that appeared in MFIRE are given in Figure 1.

```
IF() …
IF()THEN … ELSE … ENDIF
DO … CONTINUE
GOTO statements
```

**Figure 1:** Flow control statements in MFIRE

The `IF()`… and `IF()THEN`…`ELSE`…`ENDIF` statements are corresponding to the `if`…`else`… statements in C++. The `DO`…`CONTINUE` statements are equivalent to the `for` loops. Code relating to these simple structures is easily converted.

We had a lot of trouble with the `GOTO` statements in FORTRAN. The `goto` statement is also available in C++, but it is rarely seen in C++ programs because the following features are available (Figure 2).

| | |
|---|---|
| `exit` | A function that causes immediate program termination. |
| `return` | A statement that terminates the execution of a function. |
| `break` | A statement that terminates execution of a loop or switch statement. |
| `continue` | A statement that causes an immediate branch to the loop test. |

**Figure 2:** Flow control statements in C++

In our code transition, we use `while` loops and `do` … `while` loops and combined them with the statements in Figure 2 to handle all the `GOTO` statements in the FORTRAN code. We will discuss the transition of `GOTO` statements in next section.

## 4. The Code Transition

MFIRE consists of three pieces of code, `Mfire0.for`, `Mfire1.for` and `Mfire2.for`. This work is restricted in the first half of MFIRE, e.g. the code conversion of `Mfire0.for` and `Mfire1.for`.

`Mfire0.for` has just a few lines of code and has no complicated flow control structures. The conversion of code for this part has been discussed in the previous section. `Mfire1.for` has over 5000 lines of code and there are many `SUBROUTINES` and `GOTO` statements. The following subsections will discuss the conversion of the `SUBROUTINES` and `GOTO` statements.

## 4.1 The SUBROUTINE Structure

There are total 25 SUBROUTINES in Mfire1.for. They are called by the main program and other SUBROUTINES. The subroutine call relationship is given in Table 2(p.15). There are only simple calls, no recursive calls.

In conversion, the SUBROUTINES were converted to void functions in C++. The main program was converted to the main function. The DATA BLOCK was converted to a header file (var.h) where all the global variables are declared and initialized.

## 4.2 Regrouping Code into a Function

This section focuses on a common code structure we found in MFIRE. A flow chart for this structure is shown in Figure 3 (p.16). In our conversion, one function was broken down into two functions. Since the rear part of the function is executed by every branch of the code, the rear part is regrouped as a function. When the condition is true, the new function is called and returned.

Figure 5 (p.17) provides a sample piece of the FORTRAN code that was extracted from Mfire1.for. In this code segment, there are many GOTO 300 statements. The label 300 is at the end of the program. We regrouped the code between label 300 and label 350 as a new function.

The converted C++ code is listed in Figure 7 (p.18). The code between label 300 and label 350 is regrouped as a new function. Every GOTO 300 statement is converted to a function call to this new function. (m_go_to_300(marky, nstop, option, md1, j, kv)).

**Table 2:** The subroutines and their relationship in `Mfire1.for`

| main calls | Subs They call |
|---|---|
| ARR | --- |
| BASE | --- |
| CCDATA | READIN |
| CDCH | READIN |
| CDENDS | KALPHA |
| CDJUNC | --- |
| CH4EVA | --- |
| CHECK1 | --- |
| CHSFIT | LSFAN |
| | SPLINE |
| FWCT | --- |
| INPUT | READIN |
| ITR | ARR |
| | BASE |
| | LSFAN |
| | MBLNC |
| | MSLIST |
| | NVP1 |
| | NVP2 |
| | SPLINE |
| MBLNC | --- |
| MSLIST | --- |
| NVP1 | --- |
| OUTPUT | SPLINE |
| | LSFAN |
| RECIRC | CDJUNC |
| RGLT | --- |
| PREP | --- |
| TEVAL | --- |

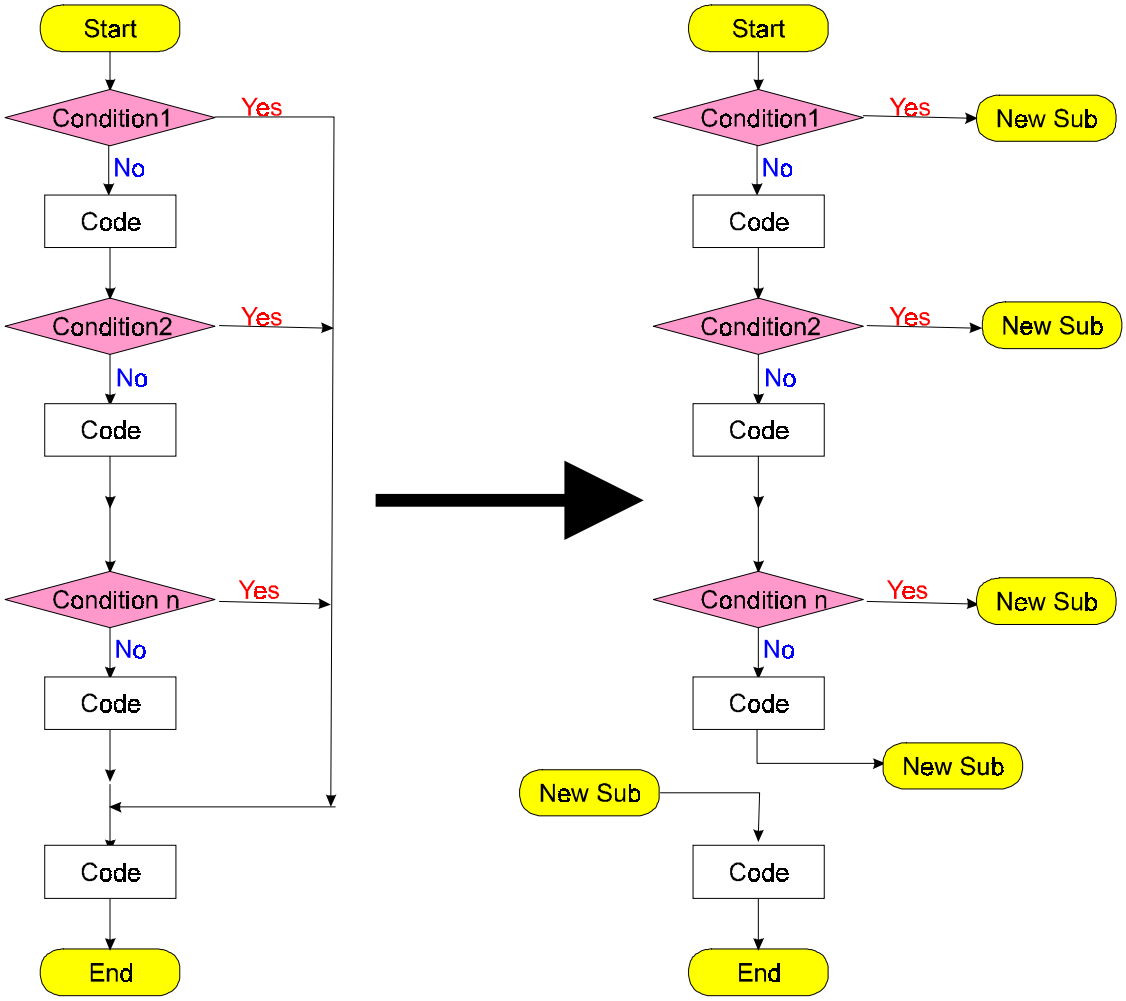


**Figure 3:** Regroup to a new subroutine or function (Flow Chart )

```
……
      CALL INPUT (1,NSTOP,MARKY,MAXNO,KV)
      IF (NSTOP.GT.0) GO TO 300
      CALL CHECK1 (NSTOP,MAXNO,KV)
      IF (NSTOP.GT.0) GO TO 300
      IF (NETW.NE.1) THEN
         CALL INPUT (2,NSTOP,MARKY,MAXNO,KV)
         IF (NSTOP.GT.0) GO TO 300
      ENDIF
……

      CALL INPUT (3,NSTOP,MARKY,MAXNO,KV)
      IF (NSTOP.GT.0) GO TO 300
      CALL CCDATA (MARKY,NSTOP)
      IF (NSTOP.GT.0) GO TO 300
      IF (MARKY.EQ.1) THEN
         CALL CDCH (NSTOP)
         IF (NSTOP.GT.0) GO TO 300
      ENDIF
……
C
      IF (NETW.GE.1.OR.NTEMP.GE.1) GO TO 300
      IF (OPTION.EQ.'CONTINUE') GO TO 350
……
C
C
300   IF (MARKY.LE.0) THEN
……

350   STOP
```

**Figure 5:** FORTRAN code to be regrouped as new function

```
……
  input (1,nstop,marky,maxno,kv);
  if (nstop > 0)
      m_go_to_300(marky,nstop,option,md1,j,kv);
  check1 (nstop,maxno,kv);
  if (nstop > 0)
       m_go_to_300(marky,nstop,option,md1,j,kv);
  assert(netw != 1);
  if (netw != 1) {
    input (2,nstop,marky,maxno,kv);
    if (nstop > 0)
         m_go_to_300(marky,nstop,option,md1,j,kv);
  }

……
  input (3,nstop,marky,maxno,kv);
  if (nstop > 0)
      m_go_to_300(marky,nstop,option,md1,j,kv);
  ccdata (marky,nstop);
  if (nstop > 0)
     m_go_to_300(marky,nstop,option,md1,j,kv);
  if (marky == 1)
    cdch (nstop);
  if (nstop > 0)
     m_go_to_300(marky,nstop,option,md1,j,kv);

……
  if (netw >= 1 || ntemp >= 1)
      m_go_to_300(marky,nstop,option,md1,j,kv);


……

void m_go_to_300(int &marky,int& nstop,
                 char*option,char*md1,
                  int&j, int&kv){
  int i=0,nnb=0,nnj=0,nnfnum=0,k=0,l=0;
……

}
```

**Figure 7:** C++ code converted from FORTRAN code in Figure 5.

### 4.3 Modify the `IF` statements

There were several pieces of code made up with `IF` statements that were structured in the flow chart shown in Figure 9, **code2** is executed when the condition is false. So, we reversed the `if` condition and include the **code2** in the new `if` block.

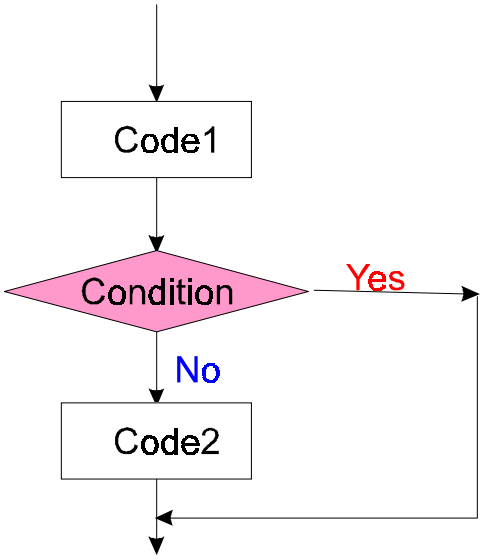


**Figure 9:** To revise the `if` statement (Flow chart)

Figure 10 is a code segment from `Mfire1.for`. Figure 12 (p.20) is the corresponding C++ code after conversion.

```
        IF (MARKY.EQ.1) GO TO 100
        NSTOP1=0
        CALL CHSFIT (NSTOP1)
100     ITCT=0
        IF (IOUT.LE.(-1).AND.MARKY.LE.0)
```

**Figure 10:** To be revised IF statement (FORTRAN)

```
if (marky != 1) {
  nstop1=0;
  chsfit (nstop1);
}
itct=0;
if (iout <= (-1) && marky <= 0)
```

**Figure 12:** Revised IF statement (C++ code after conversion)

### 4.4 Restructuring GOTO Statement into a Loop

There were several sections of code where there was a label before a block of code and an `IF` statement after it. If the condition checked by the `IF` statement was true, the `GOTO` statement would transfer control to the label before the block of code. We restructured this as a `do ... while` loop. The loop body is from the label to the `IF` statement. The loop body is to be executed at least once. Figure 15 (p.21) provides some sample code (from the function `output`) for this case. Figure 17 (p.22) is the corresponding C++ code after conversion.

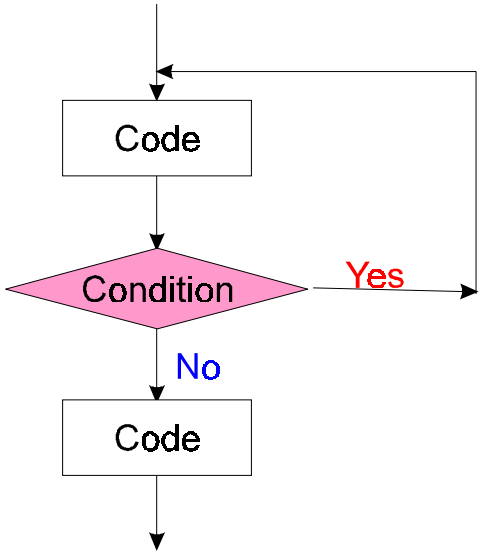


**Figure 9:** Regroup as a new `do ... while` loop (Flow chart)

```
            IZ=0
            LL=0
1530        II=0
            DO 1540 I=1,NB
               IF (NO(I).GT.0) THEN
                  IF (TQR(I).GT.TSR.OR.II.EQ.0) THEN
                     TSR=TQR(I)
                     II=I
                  ENDIF
               ENDIF
1540        CONTINUE
            IF (II.EQ.0) GO TO 1590
            IF (LL.GT.0) THEN
               KK=0
               DO 1550 J=1,LL
                  K=LST(J)
                  IF (TQR(II).GT.TQR(K)) KK=KK+1
1550           CONTINUE
               IF (KK.GE.(LL-2).AND.TQR(II).GT.20.0) GO TO 1560
               GO TO 1590
            ENDIF
1560        LL=LL+1
            LST(LL)=II
            NO(II)=-NO(II)
            JSS=JS(II)
            JFF=JF(I)
1570        DO 1580 I=1,NB
               IF (NO(I).GT.0) THEN
                  IF((JS(I).EQ.JSS.OR.JF(I).EQ.JSS.OR.JS(I).EQ.JFF.OR.
     .               JF(I).EQ.JFF).AND.(TQR(I).GT.20.0)) THEN
                     LL=LL+1
                     LST(LL)=I
                     NO(I)=-NO(I)
                     IF (JS(I).EQ.JSS) THEN
                        JSS=JF(I)
                     ELSE IF (JF(I).EQ.JSS) THEN
                        JSS=JS(I)
                     ELSE IF (JS(I).EQ.JFF) THEN
                        JFF=JF(I)
                     ELSE
                        JFF=JS(I)
                     ENDIF
                     GO TO 1570
                  ENDIF
               ENDIF
1580        CONTINUE
            IZ=IZ+1
            IF (IZ.LE.3) GO TO 1530
1590        CRT1=0.0
```

**Figure 15:** FORTRAN code to be regrouped as `do` … `while` loop

```
do {
     ii=0;
     for ( i=1;i<=nb;i++)
       if (no[i] > 0)
         if (tqr[i] > tsr || ii == 0) {
           tsr=tqr[i];
           ii=i;
         }
     if (ii==0) break;//go_to_1590;
     if (ll > 0) {
       kk=0;
       for ( j=1;j<=ll;j++){
         k=lst[j];
         if (tqr[ii] > tqr[k]) kk=kk+1;
       }
       if (kk < (ll-2) || tqr[ii] < 20.0) break;
     }
     ll=ll+1;
     lst[ll]=ii;
     no[ii]=-no[ii];
     jss=js[ii];
     jff=jf[ii];
     for ( i=1;i<=nb;i++){
       if (no[i] > 0) {
         if ((js[i] == jss || jf[i] == jss ||
               js[i] == jff ||
                  jf[i] == jff) && (tqr[i] > 20.0)) {
           ll=ll+1;
           lst[ll]=i;
           no[i]=-no[i];
           if (js[i] == jss) {
               jss=jf[i];
           }else  if (jf[i] == jss) {
               jss=js[i];
           }else  if (js[i] == jff) {
               jff=jf[i];
           }else{
               jff=js[i];
           }
           i=-1;continue;//go_to_1570;
         }
       }
     }
   iz=iz+1;
} while (iz <= 3);// go_to_1530;
crt1=0.0;
```

**Figure 17:** Regrouped as `do ... while` loop (C++ code)

The flow chart in Figure 12 (p.24) presents a more complicated situation, since there is more than one `GOTO` statement that redirects the code back to a previous position. In this case, the `GOTO` statement structure is regrouped as another type of loop. We used an additional variable to control the loop. If the loop needs to be continued, set the variable to 1, otherwise set it to 0. The `GOTO` statements are replaced with a control variable set statement and a `continue` statement.

Figure 14 (p.25) gives some sample code (from the function `ccdata`) for this case. There are two `GOTO 10` statements. The code is converted to a `while` loop. A new `int` variable (`i10`) is introduced. This variable is set to 1 before entrance into the loop. After the loop begins, the variable is set to 0. The first `GOTO 10` statement is replaced with `i10=1;` and `continue;` the second is replaced with `i10=1;`(see Figure 16, p.25).

### 4.5 Use `beak` or `continue` Instead of `GOTO` Statements in a Loop

Another common situation we encountered is where `GOTO` statements are in the middle of a loop to force the loop to end or to continue on the next iteration. We used `continue` or `break` statements at this situation, depending upon which was desired.

Figure 18 is a flow chart where a `GOTO` statement forced the end of current iteration. Figure 19 (p.26) presents some sample code (from `ch4eva` function) for this case where we used `continue` statements instead of `GOTO` statements. Figure 21 (p.27) is the corresponding C++ code after conversion.

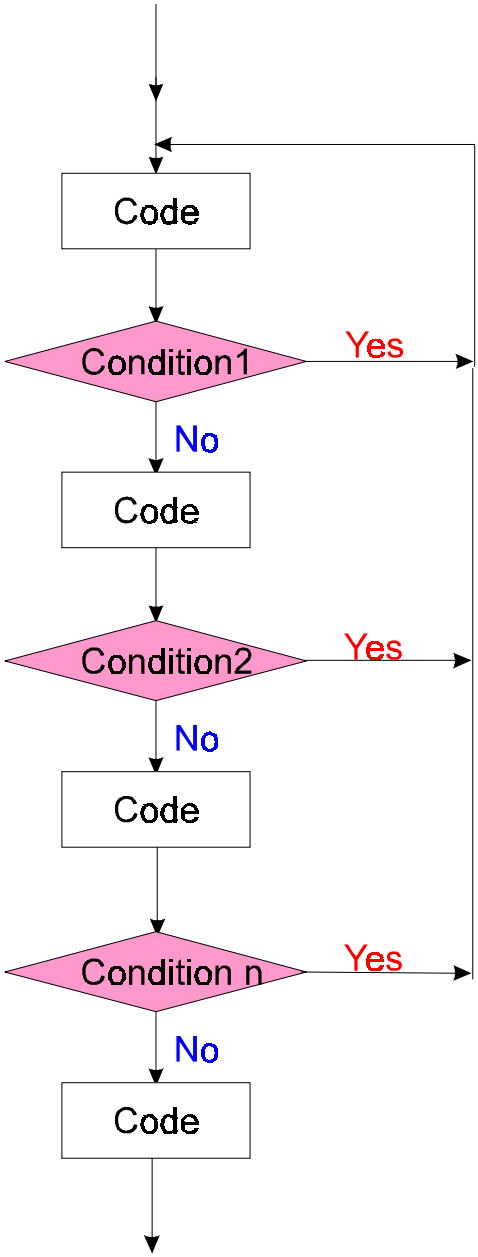


**Figure 12:** Regrouped to `while` loop (Flow Chart)

```
10        CALL READIN (DAL,6,ISTOP,0)
          IF (ISTOP.EQ.1) THEN
             WRITE (8,240)
             WRITE (8,250) (ROW(IE),IE=1,80)
             NSTOP=5
             RETURN
          ENDIF
          IF (MARKY.EQ.0) WRITE (6,260)
          IF (DAL(15).LT.(-1.E20)) GO TO 10
          IF (NCOMTS.GT.NCOMT2) THEN
             NCOMT2=NCOMTS
             GO TO 10
         ENDIF
```

**Figure 14:** Regroups to `while` loop (FORTRAN code)

```
i10=1;
while(i10){
    i10=0;
    readin (dal,6,istop,0);
    if (istop == 1) {
      assert(0);
      of8<<str240;
      of8<<str250;
      for(ie=1;ie<=80;ie++)of8<<row[ie];
      nstop=5;
      return;
    }
    if (marky == 0) cout <<str260;
    if (dal[15] < (-1.e20)){
      i10=1;
      continue;
    }
    if (ncomts > ncomt2) {
      ncomt2=ncomts;
      i10=1;
    }
}
```

**Figure 16:** Regrouped to `while` loop (C++ code).

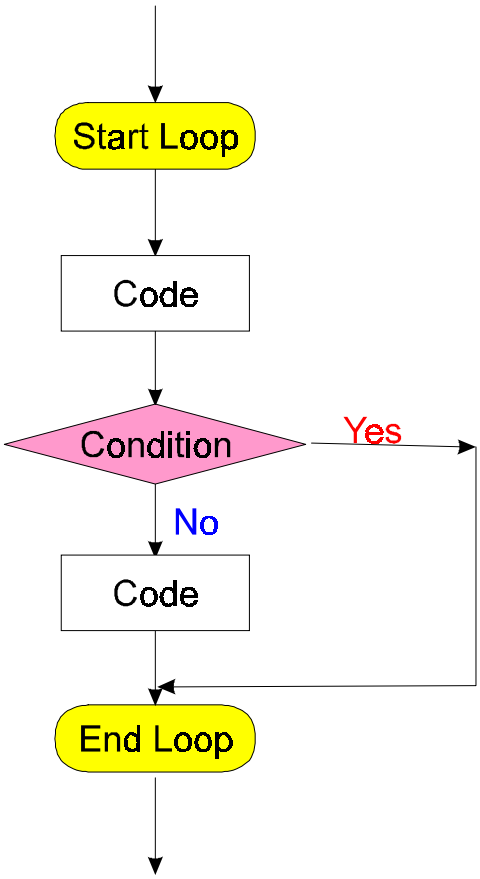


**Figure 18:** To be replaced by `continue` (Flow chart)

```
      DO 40 I=1,NB
         IF (CH4V(I).GT.0.0) GO TO 40
         CH4V(I)=CH4PA(I)*LA(I)*O(I)
         IF (CH4V(I).GT.0.0) GO TO 40
         M=0
         N=0
         DO 30 L=1,NJ
            IF (JS(I).EQ.JNO(L)) THEN
               CH4S=CH4C(L)
               M=1
            ENDIF
            IF (JF(I).EQ.JNO(L)) THEN
               CH4F=CH4C(L)
               N=1
            ENDIF
            IF (M+N.GT.1) THEN
               IF (CH4F.GT.CH4S.AND.Q(I).GT.0.0) THEN
                  ……
               ENDIF
               GO TO 40
            ENDIF
30       CONTINUE
40    CONTINUE
```

**Figure 19:** To be replaced by `continue` statement (FORTRAN code)

```
for ( i=1;i<=nb;i++){
  if (ch4v[i] > 0.0) continue;
  ch4v[i]=ch4pa[i]*la[i]*o[i];
  if (ch4v[i] > 0.0) continue;
  m=0;
  n=0;
  for ( l=1;l<=nj;l++){
    if (js[i] == jno[l]) {
   ch4s=ch4c[l];
   m=1;
    }
    if (jf[i] == jno[l]) {
   ch4f=ch4c[l];
   n=1;
    }
    if (m+n > 1) {
   if (ch4f > ch4s && q[i] > 0.0) {
     ……
   }
   break;
    }
  }
}
```

**Figure 21:** Replaced with `continue` statement (C++ code)

Figure 23 (p.28) is a flow chart of the other case that shows the `GOTO` statement forces the end of a loop. Figure 25 (p.28) presents some sample code (from `arr` function) for the case where a `break` statement is used. Figure 27 (p.29) is the corresponding C++ code after conversion.

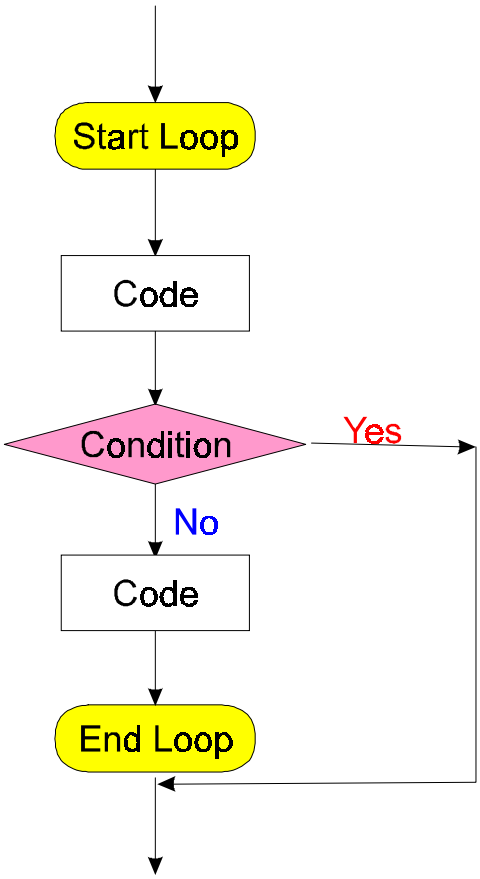


**Figure 23:** Replaced with `break` statement (Flow Chart)

```
              DO 50 J=1,NB
                 IF (NCENT(I).EQ.NO(J)) THEN
                    NWTYP(J)=10
                    GO TO 60
                 ENDIF
50            CONTINUE
60       ……..……
```

**Figure 25:** To be Replaced with `break` statement (FORTRAN code)

```
for ( j=1;j<=nb;j++)
     if (ncent[i] == no[j]){
       nwtyp[j]=10;
       break;
     }
```

**Figure 27:** Replaced with `break` statement (C++ code)

## 4.6 `GOTO` Statements in Nested Loops

If the `GOTO` statement jumps within a loop, it is the same as in the last subsection. Here we discuss two-layer nesting. We discuss the case that the inner loop ends and forces the outer loop to the next iteration. In this case, there are two different situations. There is or is not additional code between the end of the inner loop and the end of outer loop. If there is not, just replace the `GOTO` statement with a `break` statement. If there is, we added an addition `if` clause and a `continue` statement immediately after the inner loop.

Figure 29 (p.30) is a flow chart in the case that the `GOTO` statement jumps the code from the inner loop to the outer loop and there is no code between the end of the inner loop the end of the outer loop. Figure 31 (p.31) presents some sample code (from `arr` function) for the case. Figure 33 (p.31) is the C++ code after conversion one `break` statement is used.

Figure 35 (p.32) is the other situation where there is some code between end of the inner loop and the end of outer loop. Figure 37 (p.33) is the sample FORTRAN code in this case. Figure 39 (p.33) is the converted C++ code. In this case, the `GOTO` statement is replaced by a `break` statement to exit the inner loop. Since there are some code after the inner loop, an `if` clause is used after the end

of the inner loop. If the inner loop is ended abnormally, `continue` statement need to be executed to force the outer loop to the next iteration.

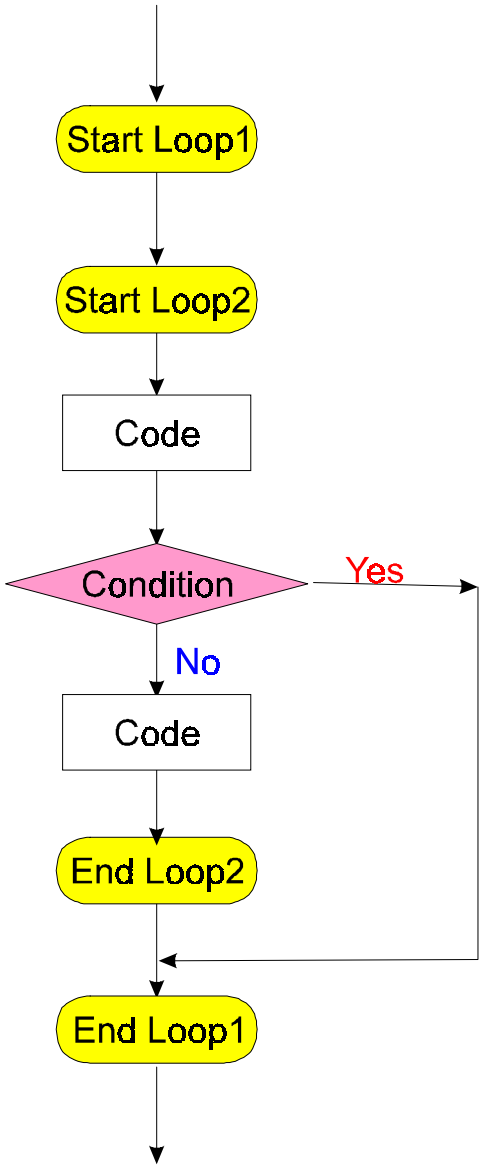


**Figure 29:** Nested loops case I (Flow chart)

```
          DO 60 I=1,INFLOW
             DO 50 J=1,NB
                IF (NCENT(I).EQ.NO(J)) THEN
                   NWTYP(J)=10
                   GO TO 60
                ENDIF
50           CONTINUE
60        CONTINUE
```

**Figure 31:** Nested loop case I (FORTRAN code)

```
 for ( i=1;i<=inflow;i++){
   for ( j=1;j<=nb;j++)
      if (ncent[i] == no[j]){
        nwtyp[j]=10;
        break;
      }
 }
```

**Figure 33:** Nested loops case I (C++ code)

### 4.7 The Combinations of Situations

In `Mfire1.for`, there are several cases where exist multiple situations discussed above. In this case, we need to be very careful with the code conversion. We can easily make an error with complicated `GOTO` statement structures. We need to regroup `GOTO` statement into a new `do … while` or `while` loops, change `GOTO` statements to `continue` or `break` statements, and/or add additional `if` clauses. Each different structure has a different solution.

Figure 41 (p.34) presents a sample flow chart where `GOTO` statement is difficult to be regrouped. Figure 43 (p.35) presents some sample code from function `itr`. There are many `GOTO` statements. Figure 45 (p.36) is the

converted C++ code. We converted `GOTO 5` and `GOTO 10` into two loops using the technique described above. Other `GOTO` statements are converted to `break` or `continue` statement. Since the structure is complicated, we spent a lot of time to get the correct code conversion.

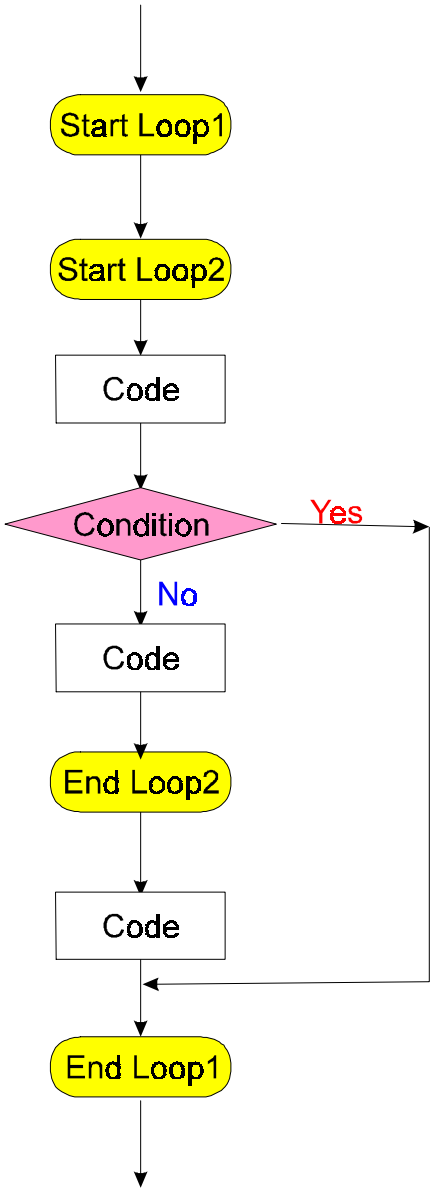


**Figure 35:** Nested loops Case II (Flow chart)

```
      DO 10 K=1,NB
         IF (NWTYP(K).LT.0) THEN
            INU(NBU)=K
            NBU=NBU-1
         ELSE IF (NWTYP(K).EQ.0.OR.NWTYP(K).EQ.10) THEN
            RQ(K)=ABS(R(K)*Q(K))
            NWTYP(K)=2
         ELSE
            IF (NFNUM.GT.0) THEN
               DO 5 J=1,NFNUM
                 IF(NOF(J).EQ.NO(K).AND.NWTYP(K).EQ.1) GO TO 10
5              CONTINUE
            ENDIF
            INU(NBL)=K
            NBL=NBL+1
         ENDIF
10    CONTINUE
```

**Figure 37:** Nested loops Case II (FORTRAN code)

```
  for ( k=1; k<=nb;k++){
    if (nwtyp[k] < 0) {
      inu[nbu]=k;
      nbu=nbu-1;
    }else if (nwtyp[k] == 0 || nwtyp[k] == 10) {
      rq[k]=fabs(r[k]*q[k]);
      nwtyp[k]=2;
    }else{
      if (nfnum > 0){
     for ( j=1;j<=nfnum;j++)
       if (nof[j] == no[k] && nwtyp[k] == 1) break;
       if(j<=nfnum)continue;
      }
      inu[nbl]=k;
      nbl=nbl+1;
    }
  }
```

**Figure 39:** Nested loops Case II (C++ code)

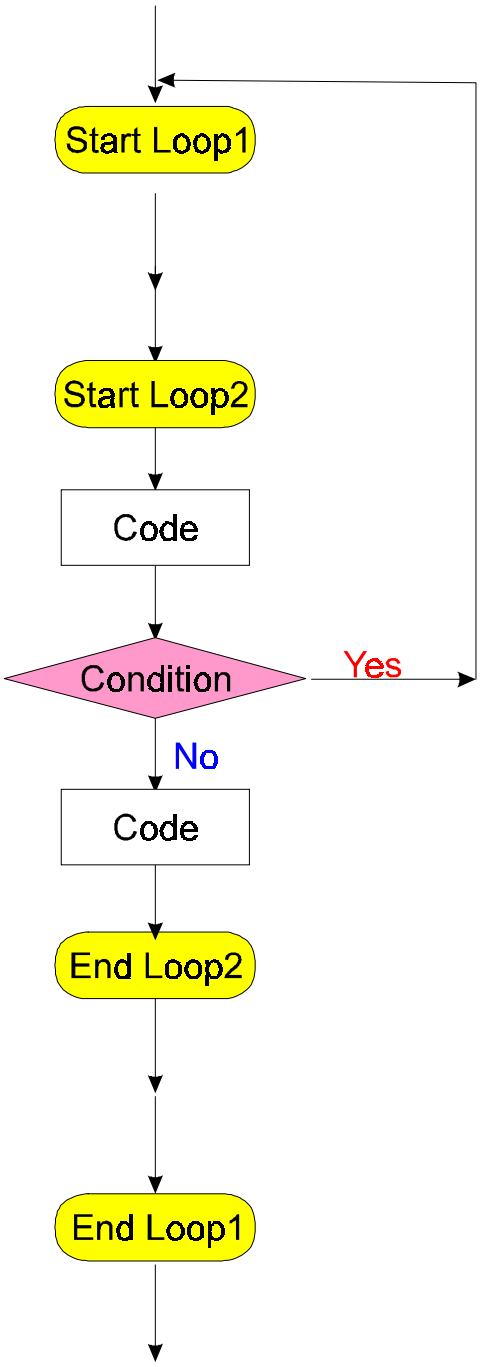


**Figure 41:** Complicated GOTO structure. A sample flow chart

```
5      IT=0
10     DQSUM=0.
       MBEGW=1
       DO 50 K=1,MNO
          ……
          IF (NWTYP(N).EQ.(-1)) THEN
             MBEGW=MENDW+1
             GO TO 50
          ENDIF
          DO 30 J=MBEGW,MENDW
             ……
             IF(NWTYP(N).EQ.1) THEN
                IF (NFNUM.GT.0) THEN
                   DO 20 L=1,NFNUM
                      IF (NFREG(L).EQ.N) THEN
                         RQSUM=RQSUM-(RGRAD(L)*100000)
                         GO TO 25
                      ENDIF
20                 CONTINUE
                ENDIF
25              IF (IABS(JS(N)).NE.JSB(N)) FACT=-FACT
                DPSUM=DPSUM-FACT*R(N)
             ELSE
                ……
             ENDIF
30        CONTINUE
          ……
          ……
50     CONTINUE
C
       DO 100 KI=1,NB
          IF (NWTYP(KI).EQ.1.AND.NFNUM.GT.0) THEN
             DO 90 J=1,NFNUM
                IF (NOF(J).EQ.NO(KI)) THEN
                   ……
                   ELSE
                      NABF=JS(KI)
                      DO 70 L=1,NJ
                         IF (NABF.EQ.JNO(L)) THEN
                            TABF=T(L)
                            GO TO 80
                         ENDIF
70                    CONTINUE
                   ENDIF
80                IF (NEGQ(J).EQ.0) THEN
                      ……
                      GO TO 100
                   ELSE
                      ……
                   ENDIF
                ENDIF
90           CONTINUE
          ENDIF
100    CONTINUE
……
          IF (IT.LE.1) THEN
             GO TO 10
          ELSE IF ((DQSUM/FLOAT(MNO)).LT.10.) THEN
             RETURN
          ELSE
             ……
             GO TO 5
          ENDIF
       GO TO 10
```

**Figure 43:** A complicated code conversion (FORTRAN). FORTRAN code from ITR SUBROUTINE. There are many complicated GOTO structures. For saving space, structure unrelated statements are omitted.

```
i5=1;while(i5){
  it=0;
  i10=1;while(i10){
    ……
    for ( k=1;k<=mno; k++){
      ……
      if (nwtyp[n] == (-1)) {
        mbegw=mendw+1;
        continue;
      }
      for ( j=mbegw;j<=mendw;j++){
        ……
        if(nwtyp[n] == 1) {
          if (nfnum > 0) {
            for ( l=1;l<=nfnum;l++)
              if (nfreg[l] == n) {
                rqsum=rqsum-(rgrad[l]*100000);
                break;
              }
          }
          if (abs(js[n]) != jsb[n]) fact=-fact;
          dpsum=dpsum-fact*r[n];
        }else{
          ……
        }
      }
      ……
    }
    for ( ki=1;ki<=nb;ki++){
      if (nwtyp[ki] == 1 && nfnum > 0) {
        for ( j=1;j<=nfnum;j++){
          if (nof[j] == no[ki]) {
            ……
            }else{
              nabf=js[ki];
              for ( l=1;l<=nj;l++){
                if (nabf == jno[l]) {
                  tabf=t[l];
                  break;
                }
              }
            }
            if (negq[j] == 0) {
              ……
              break;
            }else{
              ……
            }
          }
        }
      }
    }
    it=it+1;
    itct=itct+1;
    if (it > 1){
      continue;
    }else  if ((dqsum/double(mno)) < 10.) {
      return;
    }else  if (itct > 500) {
      ……
      break;
    }
  }
}
```

**Figure 45:** A complicated code conversion (C++ code). C++ code from `itr` function. There are many complicated GOTO structures. For saving space, structure unrelated statements are omitted.

## 5. Debugging the Code

Through the compilation process, most of the typos of the C++ code after conversion were picked out and fixed. We then were able to run the C++ code after compilation, and the result was not the same as the FORTRAN code. The differences were caused from additional typos and some logical errors. We spent approximately the same amount of time finding and fixing the errors as we spent on the code conversion.

We wrote an additional function `void prnt()` and `SUBROUTINE PRNT` to help locating errors. This function outputs all the global arrays. The output formats of C++ and FORTRAN version were done as close to the same as possible, and when this function is executed, the program exits.

The debug process begins in the main function. The `prnt`() function call is inserted immediately before a function call (the function `input`). If the output values of the arrays of C++ code match that of the FORTRAN code, the code before the insertion point is correct. If not, we know something wrong with the code up to this part. Move the `prnt`() function call backwards in order to locate the errors, then fix them. If the code before the insertion point is correct, move the `prnt`() after the next function call. If the output values of the arrays of C++ code match that of the FORTRAN code, the code of that function is correct. If not, something must be wrong in that function. Insert `prnt`() function call into the function to locate and fix errors in the function.

For loops, we have experienced that the output of `prnt`() function matches when `prnt`() in inserted at any place within the loop and does not match when `prnt`() is inserted immediately after the loop. In this case, the first iteration of the loop is correct and something is wrong with a later iteration. To locate this kind of errors, we introduced an additional integer variable to count the iterations of the

loop and insert an if clause before `prnt()` function call. Then, we could locate in which iteration the error happens and fix that error.

Now you move the `prnt()` function back to the main function and continue debugging. By applying `Testdata.dat` as input data, we fixed every error of our code. We applied other test data files to our code. Using the same technique, additional errors are detected and fixed. Our C++ code now works correctly for all the data files we have been given. The performance of our C++ code is exactly the same as the original FORTRAN code.

# 6. Conclusions and Future Work

## 6.1 Conclusions

MFIRE is a large mine ventilation system. This system has been developed over dozens of years. It is a crystal from the wisdom of many engineers and professionals. Computer languages and techniques are developing very fast. The output of MFIRE is in the form of tables of numbers and need to be interpretated by professionals. Current computer techniques allow us to virtually display the result of MFIRE as a three-dimensional scene on an screen. The goal of our virtual reality project is to realize this possibility. MFIRE was written in FORTRAN code, a computer language that was developed in the early 1950s and lacks most of the good features (such as Object orientated Programming, OOP) of modern advanced languages, such as C++. The difference between FORTRAN and C++ is very big, we modified the structure of the code. My role in this project was to convert the FORTRAN code into C++ code and make sure the performances of the code are exactly the same. We have a function to output all the values of the global arrays. This is ready for three-dimension design.

### 6.2 Future Work

This paper is the first part of our big project of virtual reality mine ventilation system. We are ready to do the following:

1. Simulation modules

Based on our C++ code, the values of arrays at each stage of calculation will be output as the input of VR interface. In our code conversion, we have a function ready for this purpose.

2. VR interface design

The VR interface consists two parts: the static mine system and the dynamic ventilation system. The static part would not change once the program begins, while the dynamic part changes according to the results of the ventilation calculation. The changes of the concentration of different type of gases will be indicated with different colors. The critical situation can then be displayed on the screen.

## References


[1] Bastow, K.R., 1979, "Real-Time Simulation of Contaminant Flow Through Mine Ventilation Networks Under the Influence of Mine Fires," *M.S. Thesis*, MI Technol. Univ., Houghton, MI.

[2] Chang, X., 1987, "Digital Simulation of Transient Mine Ventilation," *Ph.D. Thesis*, MI Technol. Univ., Houghton, MI.

[3] Chang, X., and Greuer, R.E., 1985, "Simplified Method To Calculate the Heat Transfer Between Mine Air and Mine Rock," *Proceedings of the 2nd U.S. Mine Ventilation Symposium*, ed. by P. Mousset-Jones, A.A. Balkema, pp. 429-438.

[4] Chang, X., and Greuer R.E., 1987, "A Mathematical Model for Mine Fires," *Proceedings of the 3rd U.S. Mine Ventilation Symposium. Soc. Min. Eng*., AIME, Littleton, CO, pp. 453-462.

[5] Gangal, M.K., Dainty, E.D. & Kunchur, G., 1991, "CO2 as an Exhaust Emissions Surrogate in Small Dieselized Mines," *Procs. 5th U.S. Mine Ventilation Symposium*, West Virginia, June 3-5, 1991, pp 280-287, AIME.

[6] Gangal, M.K. & Dainty, E.D., 1993, "Development of a Diesel Vehicle Operator CO2 Exposure Meter," *Procs. 6th U.S. Mine Ventilation Symposium*, Salt Lake City, Utah, June

21-23, 1993, pp 65-69, AIME.

[7] Gangal, M.K. & Pathak, J., 1992, "A Case for Conservation of Electrical Energy in Canadian Underground Mines", *Procs. 11th West Virginia University (WVU) International Mining Electrotechnology Conference*, July 29-30, 1992, Morgantown, West Virginia, pp 87-91.

[8] Greuer, R.E., 1977, "Study of Mine Fires and Mine Ventilation; Part I. Computer Simulation of Ventilation Systems Under the Influence of Mine Fires," (contract SO241032, MI Technol. Univ.) *BuMines OFR* 115(1)-78, 165 pp.; NTIS PB 288 231.

[9] Greuer, R.E., 1981, "Real-Time Precalculation of the Distribution of Combustion Products and Other Contaminants in the Ventilation System of Mines," (Contract JO285002, MI Technol. Univ.). BuMines OFR 22-82, 263 pp.; NTIS PB 82-183104.

[10] Greuer, R.E., 1983, "A Study of Precalculation of Effect of Fires on Ventilation System of Mines," (contract JO285002, MI Technol. Univ.) *BuMines OFR* 19-84, 293 pp.; NTIS PB 84-159979.

[11] Greuer, R.E., 1979, "A New Computer Program for the Design of Ventilation Emergency Plans," *Proceedings of 2nd. International Mine Ventilation Congress,* Reno, NV, pp. 129-134.

[12] Hardcastle, S.G., Gangal, M.K, Udd, E. J., Grenier, M.G. & Klinowski, G.W., 1995, "Mine Ventilation and Optimization", *Procs. 26th Int. Conf. of Safety in Mines Research Institutes,* September 1995, Katowice, Poland, Vol.II, pp183-199.

[13] Hardcastle, S.G., 1995, "3D-CANVENT: An Interactive Mine Ventilation Simulator," *Procs. 7th U.S. Mine Ventilation Symposium,* June 5-7, 1995, Lexington, Kentucky, AIME.

[14] McElroy, G.E., 1954, "A Network Analyzer for Solving Mine-Ventilation-Distribution Problems," *BuMines IC* 7704, 13 pp.

[15] Sheng, J., 1984, "Determination of the Cumulative Exhaust Effects of Diesel Powered Equipment Underground," *M.S. Thesis,* MI Technol. Univ., Houghton, MI.

[16] U.S. Bureau of Mines, 1995, "U.S. Bureau of Mines training workshop on the "MFIRE" mine fire and ventilation simulator, MFIRE users manual version 2.20," August 1995 U. S. Bureau of Mines, Twin Cities Research Center, Minneapolis, MN