Inst. Phys. Conf. Ser. No 175
Paper presented at 7th Int. Conf. Computational Accelarator Physics, Michigan, USA, 15–18 October 2002
©2004 IOP Publishing Ltd

The COSY language independent architecture: porting COSY source files

L. M. Chapin†, J. Hoefkens†§ and M. Berz†

† Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, USA

Abstract. As part of the project of porting the code COSY INFINITY[1] to a language independent architecture, a tool is being developed that translates COSY language to C++ language. The tool will allow the use of C++ classes for all elementary operations of COSY, which represent a wrapper around high-performance COSY C source. This will allow those unfamiliar with the COSY language access to the power of COSY.

1. The COSY language independent architecture and the COSY syntax

Recently the COSY system [1, 2, 3, 4, 5, 6] has been ported to a language independent platform to accommodate the use in various modern and widely used programming environments. In particular, the languages supported include FORTRAN 77, C, as well as the object oriented environments of FORTRAN 90 and C++. This is achieved in the following way. The original FORTRAN 77 based language environment consists mainly of operations on a suite of COSY objects, most notably the DA (Differential Algebra) objects that have proven very useful for beam physics purposes [7]. These objects can traditionally be utilized either within the environment of a FORTRAN precompiler[8, 9, 10], or within the object oriented language environment of the COSY scripting tool[1], which will be discussed in detail below. Support in the C world is achieved by the use of a FORTRAN-to-C precompiler, and the FORTRAN source has been adjusted to achieve reliable cross-compilation to the C environment that results in efficient code. To assure future reliability and reproducibility of this porting operation, the C source code of the cross-complier is maintained together with the FORTRAN package. The use within C++ and FORTRAN90 are achieved by a suite of light wrappers providing C++ and FORTRAN90 objects for the COSY objects [11]. Great care is taken to maintain the high performance of these data types, which is achieved by merely representing the objects via pointers into COSY's original memory management system. Altogether this approach allows the rendering of COSY objects in four different language environments, while developers need only to write and maintain the FORTRAN 77 based source code in COSY.

Besides the use of data types of COSY in other languages, it is also desirable to port high-level code written in the COSY scripting language to other languages. The COSY language is an object oriented scripting tool that supports dynamic typing. It is recursive and allows nesting, local variables, and procedures. Its syntax concepts are similar to those of PASCAL, but in addition COSY has object oriented features. All type checking is done at run time, not at compile time.

Most commands of the COSY language consist of a keyword, followed by expressions and names of variables, and terminated by a semicolon. The individual entries and the

§ Currently at GeneData AG, CH-Basel 4016, Switzerland.

semicolon are separated by blanks. The exceptions are the assignment identifier ":=", and the call to a procedure, in which case the procedure name is used instead of the keyword. Line breaks are not significant; commands can extend over several lines, and several commands can be in one line. Everything contained within a pair of curly brackets "{" and "}" is ignored. Each keyword and each name consist of up to 32 characters. The case of the letters is not significant. The basic COSY commands consist of the following commands:

```
BEGIN        END
PROCEDURE    ENDPROCEDURE
FUNCTION     ENDFUNCTION
VARIABLE
LOOP         ENDLOOP
WHILE        ENDWHILE
IF   ELSEIF  ENDIF
```

as well as the assignment command. (For more details, see p. 18 ff in [1], and p. 27 for a complete list.) In the following we describe a tool that converts code written in COSY syntax to C++.

2. Outline of the method

The conversion between COSY syntax and C++ syntax is accomplished by means of a program written in Perl [12]. The program begins by reading in all of the lines of the source .fox COSY file and placing each one, delimited by a semicolon, into an array called @Lines, with a separate entry for each line. The array is then processed and various changes are made to each line, and the order is changed in accordance with C++ standard syntax. As an example of this first type of change, a COSY loop construct "LOOP I 1 TWOND 2 ;" would be changed to the C++ equivalent "for (I = 1 ; I <= TWOND ; I+=2)".

The second type of change includes the unnesting of COSY functions and returning of corresponding C++ functions in the appropriate order. The modified contents of that array are then outputted to a new C++ file, which can then be read through and even modified. The resulting C++ source will be fully compilable and, at execution time, will exactly duplicate the performance of the original COSY source.

3. COSY intrinsic functions and procedures

All of the COSY intrinsic functions and procedures can be accessed through the COSY class. The converter program gives calls in the proper format; in COSY, calls may be made with no parentheses or commas between the arguments, but they are necessary in C++. A few of the names are slightly different (normally with different capitalization) to avoid conflict with C++ reserved words, such as "int." The renaming is based on a translation table (pp. 44-48 in [1]). For example: "SIN(X)" becomes "Cosy sin (x)" and "PROCEDURE MEMALL Y" becomes "void memall (y)".

4. COSY commands

The Programming Manual lists the keywords in COSY (p. 27 in [1]). The simple commands are reformatted with a one line statement when they are encountered. The commands that are complicated enough to need more than one line of manipulation have their own subprograms that are called when a line is found to contain a command. The line is sent to the subprogram as an argument. Each subprogram breaks the line up and takes pieces as it needs them, processes the line, and returns it to its place. For example, upon encountering a line containing the word 'WRITE' such as "WRITE 6 'Hello' ;", the program would call the following Perl subprogram.

```
sub write
{
   my($write) = $_[0] ;
   #creates a local variable for the argument sent, i.e. The line

   $write =~ s/\b&\b/+/g ;
   #changes the Cosy concatenation symbol for the C++ symbol

   $write =~ s/(?i)write 6/cout << /i ;
   #changes the Cosy output command for the C++ outstream

   $write =~ s/'/"/g ;
   #changes Cosy's single quotes around strings
   #to double quotes for C++

   $write = $write." ; \n" ; #returns the modified line
}
```

Here `$_[0]` is the default variable for the first argument sent to the subprogram in Perl (the arguments are stored as entries to the array `@_` and are accessed by `$_[0]`, `$_[1]`, `$_[2]`, etc. for each argument).

The subprograms for the other commands are generally much more complicated, but follow the same basic idea.

5. User-declared procedures and functions

In COSY, following conventional PASCAL philosophy, it is permitted to nest functions and procedures within each other infinitely deep and to even entangle the functions and procedures. For example, it's fine to declare:

```
PROCEDURE PROTEST1 X Y ;
   PROCEDURE PROTEST2 Z ;
      FUNCTION FUNTEST1 A ; FUNTEST1 := A +1 ; ENDFUNCTION ;
      Z := FUNTEST1(Z) ; ENDPROCEDURE ;
   PROTEST2 Y ; X := X+Y ; WRITE 6 X ;
   ENDPROCEDURE ;
```

In C++, functions cannot be nested, and the function signatures should be declared at the beginning of the converted program and the functions should be defined at the end. Thus, a user COSY code must be reshaped before being converted to C++. "PROCEDURE"s may be regarded as void functions for the purposes of C++. The subprograms of the conversion program that reshape the "PROCEDURE"s are very similar to those that reshape "FUNCTION"s, with the exception of being of type void and thus not returning anything. They are treated together for reshaping purposes.

(i) The conversion program reads through the array until it first finds the keyword “ENDFUNCTION” or “ENDPROCEDURE,” assigning the variable ‘toggle’ to “FUNCTION” or “PROCEDURE” depending on which is found first.

(ii) The index of the end‘toggle’ line is recorded and the search reverses and looks for the closest preceding ‘toggle,’ recording its index.

(iii) All of the array entries between the ‘toggle’ index and the end‘toggle’ index are inclusively removed from the array and concatenated into a single string (with the entries separated by semicolons “;”).

(iv) The string is passed to the conversion ‘toggle’ subprogram, which takes care of formatting details. The nested structure is preserved by appending the names of any functions or procedures the function is nested within to the end of the name.

(v) The formatted function is then stored in a separate array and the process is repeated, assigning ‘toggle’ the value of “FUNCTION” or “PROCEDURE” each time depending on the next one found, until all of the “END”‘toggle’ s are removed from the @Lines array.

(vi) Each function is then split back into separate entries by semicolons and added to the end of the @Lines array.

As a result of this process, the reshaped intermediate code of the example above would look like this:

```
FUNCTION FUNTEST1_PROTEST2_PROTEST1 A ;
            FUNTEST1 := A +1 ;
ENDFUNCTION ;

PROCEDURE PROTEST2_PROTEST1 Z ;
            Z := FUNTEST1(Z) ;
ENDPROCEDURE ;

PROCEDURE PROTEST1 X Y ;
           PROTEST2 Y ;
           X := X+Y ;
           WRITE 6 X ;
ENDPROCEDURE ;
```

The subprograms of the converting program that format the “FUNCTION”s and “PROCEDURE”s differ only slightly. The following describes the formatting ”FUNCTION” subprogram in detail.

(i) The conversion subprogram for functions takes all of the lines that were between the command “FUNCTION” and the command “ENDFUNCTION” concatenated into one line as an argument.

(ii) The line is immediately divided up into local variables for function name, arguments, and the rest of the lines.

(iii) An opening curly bracket is appended to the arguments. If there is more than one argument, commas are placed between them. The set of arguments is placed in parentheses and each argument is typed as ‘Cosy&’. This is a reference to the actual variable. An ‘f’ for function or a ‘p’ for procedure is placed at the beginning of the name to establish uniqueness. Thus, for example, " FUNCTION FUNTEST1_PROTEST2_PROTEST1 A " in COSY has become "Cosy fFUNTEST1_PROTEST2_PROTEST1 (Cosy& A)".

(iv) A COSY variable with the same name as the function is declared, preceded by an "i", because, in COSY, functions deal with and return such a variable, which is declared only in naming the function. "Cosy iFUNTEST1_PROTEST2_PROTEST1 (1)," is the return variable for the above example.

(v) The rest of the lines are returned to the function and the final curly bracket closes it.

(vi) The names of any variables declared in the function (including the name variable and arguments) are placed in a string that becomes the "namespace" of the function. The "namespaces" that are visible to the function are declared using "using namespace" [13]. See the section on variables below.

(vii) The entire array @Lines holding the rest of the program is then queried for calls to the function, which are reformatted to call " $functionName" with parentheses surrounding and commas separating the arguments and the full function name.

(viii) All keywords are searched for and dealt with by their individual subprograms.

(ix) A function signature is also generated to be placed at the beginning of the output program so that the function may be placed at the end of the main output program in the style of C++. The function is then returned to the @Lines array.

The resulting C++ code for this example would be:

```
Cosy fFUNTEST1_PROTEST2_PROTEST1 (Cosy& A) ;
void pPROTEST2_PROTEST1 (Cosy& Z) ;
void pPROTEST1 (Cosy& X, Cosy& Y) ;

namespace PROTEST1
{
         Cosy X (1) ;
         Cosy Y (1) ;
}

namespace PROTEST2_PROTEST1
{
         Cosy Z (1) ;
}

namespace FUNTEST1_PROTEST2_PROTEST1
{
         Cosy iFUNTEST1_PROTEST2_PROTEST1 (1) ;
         Cosy A (1) ;
}

Cosy fFUNTEST1_PROTEST2_PROTEST1 (Cosy& A)
{
         using namespace PROTEST1 ;
         using namespace PROTEST2_PROTEST1 ;
         using namespace FUNTEST1_PROTEST2_PROTEST1 ;
         iFUNTEST1_PROTEST2_PROTEST1 = A +1 ;
```

```
        return iFUNTEST1_PROTEST2_PROTEST1 ;
}

void pPROTEST2_PROTEST1 (Cosy& Z)
{
        using namespace PROTEST1 ;
        using namespace PROTEST2_PROTEST1 ;
        Z = fFUNTEST1_PROTEST2_PROTEST1 (Z) ;
}

void pPROTEST1 (Cosy& X, Cosy& Y)
{
        using namespace PROTEST1 ;
        pPROTEST2_PROTEST1 (Y) ;
        X = X+Y ;
        cout << X ;
}
```

6. Variables

COSY variables have no declared types. They can be declared with an unlimited number of arguments that will simply increase the dimensionality of the variable (while defining the dimensions).

In treating the variables of COSY programs, most of the variables could be given the type "Cosy" from the COSY class. However, many functions explicitly require integers, and the C++ version of a COSY array is based on an integer array. For these functions, it must be determined if a numerical integer or a COSY variable is passed (both equally valid options in COSY, but not acceptable in C++). In the case of a COSY variable, the variable must be cast into an integer. This is done by using the COSY class's "toDouble" Cosy to double casting function [1] and casting the resulting double as an integer in the normal C++ fashion.

In consideration of scope, COSY variables are visible inside the program segment in which they are declared as well as in all other program segments inside it. If a variable has the same name as one visible from a higher level, its name and properties override the name and properties of the higher level variable of the same name for the remainder of the block [1]. This priority is being dealt with in C++ by creating a "namespace"[13] for each function and procedure and giving access to the "namespace" in order of priority to all of the appropriate program segments. The C++ command "using namespace"[13] overrides variables in the same manner as COSY priority. The rule for granting access is that: if a "namespace"'s full name (with the nesting extensions) appears in the name of the block, the block has the right to that "namespace". This "namespace" convention is still being implemented, but appears to be a promising solution that allows for variables to maintain their short, original names while preserving scope and avoiding a mass of global variables. In COSY, the actual variable is passed to a function and not just a copy, as is common in C++. To account for this, all variables are passed by reference to functions in the converted code [13].

7. Array access

To access arrays conveniently using all of the arguments used to call an array in COSY, two C++ functions were written that are appended to the end of the converted program and used throughout : "GetIt" and "SetIt". These functions take advantage of member access functions of the COSY class [1]. The `[]` access is not implemented in the COSY class and so is not presently available in the conversion.

8. Examples

To provide a more comprehensive example of the conversion process, we list here the results of the translation of an entire procedure from the code COSY.FOX, a part of the beam physics environment of COSY [2]. The procedure, as coded in COSY language, has the following form:

```
PROCEDURE DL L ;                     {FIELD-FREE DRIFT}
   VARIABLE I 1 ; VARIABLE J 1 ;
   IF LUM#1 ; WRITE 6 ' *** ERROR, CALL UM BEFORE ELEMENT' ;
      QUIT 0 ; ENDIF ;
   DLACT MAP L ; IF NRAY#0 ; UMS ; DLACT MSC L ; ENDIF ;
   LOCSET 0 L 0 L 0 0 0 ; IF CONS(L)>0 ; DR := 0 ; ENDIF ;
   LOOP I 1 3 ; LOOP J 1 3 ; SSCR(I,J) := 0*DD(1) ; ENDLOOP ;
   SSCR(I,I) := 1+0*DD(1) ; ENDLOOP ; UPDATE 0 1 1 ;
   ENDPROCEDURE ;
```

After conversion, the C++ code will have the following form:

```
void pDL (Cosy& L) ;

namespace DL
{
          Cosy J (1) ;
          Cosy I (1) ;
          Cosy L (1) ;
}

void pDL ( Cosy& L)
{
           using namespace DL ;
           /*FIELD-FREE DRIFT*/
           if( LUM!=1 )
           {
                      cout <<  " *** ERROR, CALL UM BEFORE ELEMENT "  ;
                      break 0  ;
           }
           DLACT ( MAP, L )  ;
           if( NRAY!=0 )
           {
                      UMS()  ;
```

```
                DLACT ( MSC, L )  ;
        }
        LOCSET ( 0, L, 0, L, 0, 0, 0 )  ;
        if( cons(L)>0 )
        {
                DR = 0  ;
        }
        for ( I = 1  ; I <= 3 ; I++)
        {
                for ( J = 1  ; J <= 3 ; J++)
                {
                        SetIt( 0*GetIt(DD, 1 ) , SSCR , I, J )  ;
                }
                SetIt( 1+0*GetIt(DD, 1 ) , SSCR , I, I )  ;
        }
        UPDATE ( 0, 1, 1 )  ;
}
```

We note that for the purpose of readability, the procedures, functions, and variables declared elsewhere and found in this procedure were translated with the procedure and then removed. For example, LOCSET is not declared as a procedure here, but earlier in the code COSY.FOX. If one tried to translate just this excerpt, the conversion program would not have identified LOCSET as a procedure and would not have processed it as such, adding the parentheses and commas.

Acknowledgments

The work was supported by the US Department of Energy and the National Science Foundation. We would like to thank K. Makino for various useful comments.

References

[1] M. Berz, J. Hoefkens, and K. Makino. COSY INFINITY Version 8.1 - programming manual. Technical Report MSUHEP-20703, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2001. See also http://cosy.pa.msu.edu.

[2] M. Berz and K. Makino. COSY INFINITY Version 8.1 - user's guide and reference manual. Technical Report MSUHEP-20704, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2001. See also http://cosy.pa.msu.edu.

[3] K. Makino and M. Berz. COSY INFINITY version 8. *Nuclear Instruments and Methods*, A427:338–343, 1999.

[4] M. Berz, G. Hoffstätter, W. Wan, K. Shamseddine, and K. Makino. COSY INFINITY and its applications to nonlinear dynamics. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 363–365, Philadelphia, 1996. SIAM.

[5] K. Makino and M. Berz. COSY INFINITY Version 7. *AIP CP*, 391:253, 1996.

[6] M. Berz. COSY INFINITY Version 6. In *M. Berz, S. Martin and K. Ziegler (Eds.), Proc. Nonlinear Effects in Accelerators*, page 125, London, 1992. IOP Publishing.

[7] M. Berz. *Modern Map Methods in Particle Beam Physics*. Academic Press, San Diego, 1999. Also available at http://bt.pa.msu.edu/pub.

[8] M. Berz. The Differential algebra FORTRAN precompiler DAFOR. Technical Report AT-3:TN-87-32, Los Alamos National Laboratory, 1987.

[9] M. Berz. The DA precompiler DAFOR. Technical report, Lawrence Berkeley Laboratory, Berkeley, CA, 1990.

[10] M. Berz. Differential algebra precompiler version 3 reference manual. Technical Report MSUCL-755, Michigan State University, East Lansing, MI 48824, 1990.

[11] J. Hoefkens. *Verified Methods for Differential Algebraic Equations*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 2001.

[12] E. Siever and S. Spainhour and N. Patwardhad. PERL in a Nutshell : A Desktop Quick Reference. O'Reilly & Associates, Inc., CA, 1999.

[13] The C++ Resources Network, http://www.cplusplus.com, 2000-2003.