# A Fortran-95 implementation of EMTP algorithms

Jean Mahseredjian[1], Benoît Bressac[2], Alain Xémard[2], Luc Gérin-Lajoie[3], Pierre-Jean Lagacé[4]

(1)IREQ / Hydro-Québec
1800 Lionel-Boulet
Varennes, Québec,
Canada, J3X 1S1

(2)Électricité de France
Direction des Études et Recherches
1 ave. du Général de Gaulle
92141 Clamart Cedex

(3) TransÉnergie,
Hydro-Québec, Complexe Desjardins,
CP 10000, Montréal,
Canada, H5B 1H7

(4)École de Technologie Supérieure
1100 Notre Dame Ouest
Montréal,
Canada, H3C 1K3

**Abstract** - This paper presents a complete Fortran-95 based implementation of EMTP type algorithms. It demonstrates software engineering advantages and proposes formulations and programming designs most suitable for transient analysis computations in Fortran-95. Computing performance is compared against Fortran-77 usage. The coding experience presented in this paper demonstrates that Fortran-95 is not just a logical upgrade from Fortran-77, it is a modern and powerful language and should be even considered as the preferred language choice for large scale transient analysis application development.

**Keywords**: Software engineering, EMTP, Fortran

## I. INTRODUCTION

This paper presents an experience in software engineering for programming EMTP (Electromagnetic Transients Program) [1] type algorithms. Although the presented material is related to transients, offered ideas and experiments are applicable to other power system analysis applications.

There are several software engineering considerations in recoding or writing from scratch a large scale power system application. The most important consideration is the choice of the programming language. The programming language plays a predominant role in the software engineering cycle. An advanced language can provide means for fast translation of mathematical formulations into actual code. The ultimate combination is very high-level programming resulting in extremely efficient code. Although interpreter based languages (such as [2]) or visual design environments can become very powerful, they are still unable to achieve the computational speed provided by compiler based languages. Automatic compiler based code generation from high-level languages, lacks specialization and has difficulties replicating conceptual thinking for the most efficient coding path.

Increasing problem complexity requires more powerful constructs and syntax in a computer language. In the past twenty years, computer science has progressed dramatically and spurred interest on the use of new programming languages such as PASCAL, ADA and most notably C/C++ and now Java.

Large scale transient analysis applications have been traditionally coded and maintained using almost exclusively the Fortran-77 language. Fortran-77 has been standardized in 1978. Its dominance in all fields of numerical computations has kept increasing as new and highly optimizing compilers became available. For numerical and historical reasons Fortran is still considered as the language of computational science and kept on surviving in the midst of new and modern languages.

Although several modern language concepts and constructs can be imitated in Fortran-77, it remains a strongly handicapped language in several aspects and more fundamentally in dynamic data allocation, parallelism and programming safety. Fortran-77 is not an object-oriented language, it is behind C and C++ for data abstraction and it lacks recursion and data structures.

It requires a major investment and strong justifications to rewrite an existing large scale Fotran-77 application with a modern and significantly different computer language, such as C/C++. Automatic translators are unable to maintain the original conceptual thinking and usually create code maintenance problems. Experience has shown that manual conversion of Fortran legacy code into C/C++ is a extensive task, there are also concerns about reported poor performance.

Although some of Fortran-77 features are becoming obsolescent and may completely disappear in a future language revision cycle, the new Fortran-95 [3][4] standard stays fully backward compatible with Fortran-77. Fortran-95 is also a modernized language most suitable for numerically intensive applications. These are the main reasons why a natural and cost effective transition from Fortran-77 is Fortran-95. Nevertheless, the coding experience presented in this paper demonstrates that Fortran-95 is not just a logical upgrade, it is a modern and powerful language and should be even considered as the preferred language choice for power system transient analysis application development. Additionally, there is also a migration path to parallel computers, since High Performance Fortran is also based on Fortran-95.

## II. FORTRAN-95 ADVANTAGES

The most useful new features of Fortran-95 for EMTP algorithms are grouped into “array building and manipulation” and “data abstraction”. The first group includes dynamic memory allocation, data parallelism, array sections and array operations. The second group includes derived data types, programming with modules (hiding, scope and encapsulation), interface definition and operator or function overloading.

For the rest of this paper, the Fotran-95 version of EMTP will be referred to as EMTP-F95.

### A. Modularity

There are several programming language features that are useful in the programming and maintenance of large codes. Modularity is among the most important features.

The code that modifies data for a component, must be localized and confined to a related location in the program, and not spread throughout the entire code. This is what is meant by encapsulation. It goes beyond the definition of function or subroutine by allowing all related operations (methods) to be grouped around a defined data type. It shall be allowed to hide data and methods. Fotran-95 allows encapsulation and information hiding. Fortran-95 has the notion of scope. Scope means that an internal function (`FUNCTION` or `SUBROUTINE`) is within its host's scope and therefore has access to all the host's entities with the ability to call other internally defined functions. External functions can be called from anywhere and contain internal procedures.

Fortran-95 has introduced the module unit. It is used for data encapsulation and global data. The generic form of a module is

```
MODULE module-name
[specification construct]
[ CONTAINS ]
[subprogram]
END [MODULE [module-name] ]
```

Language keywords are in bold (in this paper) characters to enhance visualization. A module can be accessed by another program unit using

```
USE module-name
```

Modules permit specifying private and public attributes for data and functions. An interface block can be used to provide an explicit interface to module procedures. Functions placed in a module are automatically checked for the number of arguments and for argument types by the compiler. It is a very important safety feature in Fortran-95.

To be more specific, all EMTP components can be contained in separate and completely detachable modules. The list of components includes network element models and program procedures for network solutions and input/output operations.

Module usage dictates the design of EMTP-F95 code. A completely modular architecture is created by using a core code capable of handling all the required solution methods and interacting with network components (models) only through a specific set of communication protocols, called request signals. Components react by sending back participation data. Each component module is completely encapsulated and based on “case” selectors. Each case being a response section for the received request.

Modules are also very useful in the development project of a large scale application; programmers can work in parallel for coding separate modules by strictly defined interfacing with the core code only.

### B. Derived types

Encapsulation and information hiding allows defining derived (abstract) data types. A derived type is a user defined type built up from intrinsic types and previously defined derived types. There are several advantages in using derived data types, those that have been exploited in EMTP-F95 are code readability, simplified access to data and memory management.

The following lines of code, extracted from the Zinc Oxide ($Z_nO$) arrester model, are presented to illustrate some of the above concepts.

```
MODULE zno_branch
  USE input_data
  USE plot_memory  !to transmit outputs
  USE service
  USE sparse_main_mat, &
      ONLY: put, putnonl, fill, &
      Vaug, Vaug_c, n_Ynonlin, Inonlin
  USE all_purpose, ONLY: int2str,angle
  INCLUDE 'default_header.f90'
TYPE, PRIVATE:: zno_str
 REAL(krealhp)  :: Rss    ! steady-state R
 REAL(krealhp)  :: Vref   ! reference voltage
 REAL(krealhp)  :: Vflash ! flashover voltage
 INTEGER        :: loc    ! locator in char
 INTEGER        :: knode  ! left node vector
 INTEGER        :: mnode  ! right node vector
 REAL(krealhp)  :: Iq=zero  !Norton current source
 REAL(krealhp)  :: G=zero   !Norion admittance
 REAL(krealhp)  :: i        !element current

 REAL(krealhp)  :: vkm=zero !voltage
 INTEGER :: state=0  !flashover state
 INTEGER :: segnow=0  !operating segment at t
 INTEGER :: lastseg=0 !operating segment at t-Dt

 REAL(krealhp) :: tol=1e-6_krealhp;
END TYPE zno_str

TYPE, PRIVATE:: zno_char
 REAL(krealhp),  POINTER,DIMENSION(:) :: p  !ZnO p
 REAL(krealhp),  POINTER,DIMENSION(:) :: q !Zno q
 REAL(krealhp),  POINTER,DIMENSION(:) :: V !min
 REAL(krealhp),  POINTER,DIMENSION(:) :: vpast
END TYPE zno_char
                ……
TYPE(zno_str),  &
    DIMENSION(:), ALLOCATABLE, PRIVATE :: Zno
TYPE(zno_char), &
    DIMENSION(:), ALLOCATABLE, PRIVATE :: Znoch
                ……
CONTAINS
  SUBROUTINE znomod(ido)
   INTEGER      :: k,i,j,jj
   CHARACTER(LEN=*) ido
   REAL(krealhp) :: vnew,iguess,vguess
          ……
   todocall : SELECT CASE (ido)
   CASE('initialize') !* initialization procs
          ……
   CASE('put_nodes_in_Yaug') !*symbolic data
          ……
   CASE DEFAULT
   RETURN
   END SELECT todocall
   RETURN
  END SUBROUTINE znomod
END MODULE zno_branch
```

The following lines are explanatory remarks for the above code sample.

The chosen convention in EMTP-F95 is to use upper case only letters for language keywords. Code variables are based on lower case letters, but it is allowed to use capital letters for increased visibility of some variables, such as the data holder structures. Fortran-95 is not case-sensitive by itself. Lines or inline sections starting with an exclamation mark indicate a comment. The character & is the continuation character.

The **USE** statement requests visibility within the module's scope of public names (data and functions) available in external modules. Here the $Z_nO$ model is using data and functions from several modules. It is allowed to limit accessibility using the **ONLY** statement, as is the case for the `sparse_main_mat` module usage. In this example, only the `put`, `putnonl` and `fill` methods (functions), and only the variables `Vaug`, `Vaug_c`, `n_Ynonlin` and `Inonlin` can be accessed from `zno_branch`. Limited accessibility limits global memory usage and provides increased modularity.

A derived type (structure) is created by the type declaration keyword **TYPE**. It is allowed to initialize data fields in the derived type creator. Initialization is either a number or a previously defined variable. This component is designed using 3 structures: `Zno`, `Znoch` and `Znopar` (not shown). Since `Zno` and `Znopar` must hold data for all arresters in the solved case, it is necessary to declare them as **ALLOCATABLE**. Memory allocation issues are discussed in a following paragraph.

The **PRIVATE** keyword hides data and functions from other modules capable of connecting to this module through a **USE** statement. Procedures in other modules can never impact on declared private parts.

Fortran-95 allows controlling precision using predefined variables in the declaration statement. In the above real number declarations `krealhp` indicates the highest precision available. It is defined in the `default_precision` module. A single line of code change is thus needed to change precision in the entire program. The module `default_precision` is not explicitly referenced in the `zno_branch`. That is due to the fact that using a module provides automatic access to the modules it is using. In this case `default_precision` is used by `out_saver`, which is used by `input_data`. The **USE** feature is tricky. Chained **USE** statements can create loops. They can also sophisticate relations between modules even when carefully designed.

The core code can only access the subroutine `znomod` through a call using a request keyword. Each request keyword is handled in a separate **CASE** section when the component must participate (reply) to that request.

### C. Memory allocation

Fortran-95 alleviates a major limitation of Fortran-77 by allowing standard memory allocations and deallocation calls. An allocatable vector (or array) must be declared with using the **ALLOCATABLE** keyword. Fortran-95 also allows dynamic arrays. Dynamic arrays are used only inside the procedure, they are created on procedure entry and destroyed on procedure exit. Arrays in Fortran-95 can be viewed as objects with data and size information. Subroutine variables are defaulted to automatic.

To allocate the entire `Zno` structure for `n` arresters in one statement, it is required to use:

```
ALLOCATE(Zno(n),  STAT=j); %j flags memory error
```

To access the fields of `Zno` it is needed to use the `%` symbol, such as:

```
Zno(i)%vkm=vguess;
```

A structure can also contain separately allocatable fields, in Fortran-95 they must be declared with the **POINTER** attribute. This is useful, when, as in `Znoch`, the fields of the derived data type are not of the same length. The choice of structures and related fields is motivated by code vectorization possibilities available for their usage.

### D. Overloading

Some component case sections must be designed to reply to the core code using predefined functions available from the core code. Here are, for example, the **CASE** sections used by the RLC component for sending its equations into the core code's system of equations:

```
CASE('put_in_Yn_ss')
  RLC%gz=RLC%R+jz*(w*RLC%L- RLC%C/w); !jz=sqrt(-1)
  RLC%gz=inv_vector(RLC%gz); !1/RLC%gz, 0 when 0
  DO k=1,SIZE(RLC)
    CALL fill(RLC(k)%knode,RLC(k)%knode,RLC(k)%gz)
    ……
  END DO
    ……
CASE('put_in_Yn')
  DO k=1,SIZE(RLC)
    CALL fill(RLC(k)%knode,RLC(k)%knode,RLC(k)%g)
          ……
  END DO
```

The `fill` function transmits required data into the large system matrix. It is an overloaded function, capable of handling both complex (for steady-state) and real (for time-domain) systems of equations. Function overloading refers to using the same function name, but performing different operations based on argument type. In addition to function overloading it is allowed to define operator overloading. These are new and powerful coding options in Fortran-95, since Fortran-77 allowed overloading only for intrinsic functions and data types. To construct a generic `fill` function it is necessary to place an interface statement in the `sparse_main_mat` module:

```
INTERFACE fill
     MODULE PROCEDURE fill_c,fill_r
END INTERFACE
```

Interface functions `fill_c` and `fill_r` are separately defined. The compiler automatically calls `fill_r` or `fill_c` depending on the type of the third argument in `fill` usage, real or complex respectively. A similar design is applicable to the function `inv_vector` used in the above RLC example.

In Fortran-95 it is required to provide the overloading function (method) for all possible variations in input arguments. If a method is defined for a derived type used as a vector, it must be also defined for the same derived type used as a scalar. When scalar and vector combinations are considered and a total of na arguments are used, a total of $2^{na}$ methods must be defined. This is true even if sometimes the underlying codes can be identical. The best demonstration of this statement is given by the `angle` function used for finding the angle of a complex number or vector. Its interface is defined in the EMTP-F95 `all_purpose` module:

```
INTERFACE angle
     MODULE PROCEDURE angle_real,angle_vector
END INTERFACE
```

The function `angle_real` is given by:

```
FUNCTION angle_real(phasor)
 COMPLEX(krealhp) :: phasor
 REAL(krealhp) :: angle_real
 angle_real=&
 ATAN2(AIMAG(phasor),REAL(phasor))*rad2deg;
END FUNCTION angle_real
```

The function `angle_vector` is given by:

```
FUNCTION angle_vector(phasor)
 COMPLEX(krealhp), DIMENSION(:) :: phasor
 REAL(krealhp),&
 DIMENSION(SIZE(phasor)):: angle_vector
 angle_vector=&
 ATAN2(AIMAG(phasor),REAL(phasor))*rad2deg;
END FUNCTION angle_vector
```

The only difference between `angle_real` and `angle_vector` is in the declaration of arguments. It is achieved by noticing that in Fortran-95 intrinsic functions are overloaded to handle arrays. Some other noteworthy features appearing in the above example are: the variable `rad2deg` is obtained from the scope of the module `all_purpose`; the intrinsic **`SIZE`** function allows declaring data of the same size as the input argument.

### E. Vectorization and high-level coding

Vectorization is another key ingredient for high-level constructs. Native operators and functions in Fortran-95 are readily overloaded for handling vectors and matrices. Such overloading and the ability to access array sections, provides means for vectorized and high-level coding in EMTP-F95. Some examples are given below.

The first statement for `RLC%gz` in the above RLC example is able to act on all RLC branches through a single line of code. DO loops can be avoided in these cases. More sophistication is apparent in the computation of RLC branch current at a given time-point:

```
RLC%i_at_t=RLC%g* &
          (Vaug(RLC%knode)-Vaug(RLC%mnode))+RLC%h
```

The vector `Vaug` is the solution vector, it contains node voltages for the entire system. Using `Vaug(RLC%knode)` finds the vector of all left node voltages for all RLC branches only.

High-level coding in EMTP requires advanced functions for searching in arrays. Fortran-95 has built-in functions for testing conditions on arrays. The statements below are extracted from the ideal switch code.

```
WHERE( (Sw0%newstatus==0) .AND. (Sw0%status==1) )
      !test current for this condition
  WHERE( (ABS(Vaug(iloc+Sw0%in)) <Sw0%eps ) .OR. &
         (Vaug(iloc+Sw0%in)*Sw0%ilast < zero) )
     Sw0%newstatus=0; !allowed to open
  ELSEWHERE
     Sw0%newstatus=1; !stays closed
  END WHERE
END WHERE
IF (ANY(Sw0%newstatus.NE.Sw0%status)) THEN
          rebuild_for_sw=1; rebuild_this_sw=1;
ENDIF
```

The **`WHERE`** construct allows performing operations on selected array sections. Here it is used to test and set switch status for all switches. Using **`ANY`** provides a high-level syntax for setting the refactorization signal.

Another example of high-level coding with arrays is the usage of locator and extractor functions. In this example it is desired to find the operating segment in the nonlinear arrester model for a given voltage condition. Here is how it is expressed in Fortran-95

```
j=Zno(i)%loc+Zno(i)%state;
d=MAXLOC(Znoch(j)%V,&
         MASK=ABS(Zno(i)%vkm)>Znoch(j)%V);
Zno(i)%segnow=d(1);
```

The **`MAXLOC`** statement uses a conditional mask for locating the operating segment in a characteristic vector situated by the pointer `j`.

The solution method for nonlinear branches in EMTP-F95 is an iterative process where each nonlinear component is represented by a Norton equivalent. To avoid numerical problems it is necessary to save and refresh the sections of the sparse matrix `Yaug` where nonlinear branches are connected. This is achieved in a single statement in the code lines shown below. Since the original size of `Ynonlin` (the memory refresh matrix) is overallocated, it is necessary to pick-up the maximum number of cells using the actual size given by `n_Ynonlin`.

```
Yaug(Ynonlin(1:n_Ynonlin)%j)%value= &
Ynonlin(1:n_Ynonlin)%value;
Inonlin=zero; VaugOld=Vaug; !reset
```

The last line of this code is for resetting the entire vector `Inonlin` and saving the previous iteration solution in `Vaugold`.

### F. Object oriented programming

Fortran-95 is not a fully object oriented language. It is however feasible and practical to adopt OOP (Object Oriented Programming) in Fortran-95. Almost all features of C++ can be reproduced directly or emulated in Fortran-95. Fortran-95 has the notion of class through modules. It supports inheritance since derived types can be made of other types and create a derived class built upon the base class methods. Fortran-95 is directly capable of static polymorphism using the **`INTERFACE`** block construct. The most significant exception is runtime polymorphism (dynamic dispatching) since its emulation requires more effort.

It is not obvious to decide on how to improve productivity through OOP. Experiments demonstrated that full OOP for this type of application can create a cryptic code and drastically lack performance. That is why, even though many of OOP ideas were retained, the OOP programming paradigm was not adopted.

## III. EMTP-F95 CODING EXPERIENCE

The EMTP-F95 program used in this paper has most of the main components and solution methods of the standard EMTP. It has been entirely coded in Fortran-95. Several iterations were required before deciding on programming rules and the most appropriate selections of constructs. Those iterations allowed to experiment with high-level programming in relation with computer timings. Since overall program readability and maintainability are key factors, the final selections were made following a balance between readability and efficiency. Surprisingly, most of

the new constructs retained in Fortran-95 did not deteriorate speed. Compared to Fortran-77 and C/C++, the experience related in this paper, concludes that the created code allows a much faster software engineering cycle, is easier to follow and is more robust.

### A. Main considerations

As explained earlier, the data encapsulation and hiding principles allow coding an entirely modular software. In addition to the fact that modules can be coded in parallel, multiple authors can also develop program pieces which will be used by others. EMTP-F95 is a completely modular program. Its main architecture is composed of a core code surrounded by component modules.

The core code provides all the "methods" for handling input, output and sparse matrix solutions. Its main objective is to solve a main sparse system of the type $\mathbf{Ax} = \mathbf{B}$ or more specifically stated in the nodal analysis context, $\mathbf{Y}_{aug}\mathbf{V}_{aug} = \mathbf{I}_{aug}$. The subscript "aug" means augmented, since this formulation does not restrict to pure nodal analysis and allows including source, dependent branch and switch equations. Even though the majority of unknowns is composed of node voltages, the unknowns vector $\mathbf{V}_{aug}$ can also hold currents for the additional equations created in $\mathbf{Y}_{aug}$. More details on this formulation can be found in [5]. Its resemblances with the Tableau Approach make it most appropriate for the fully modular architecture proposed in this paper. Components can interact with the main system of equations through predefined `put` (for symbolic factorization) and `fill` (for actual values) functions.

Component modules include several services for solution methods and all the network component models. Each EMTP-F95 model is coded in a separate and completely detachable module. In fact a module can be ultimately rendered as a DLL (Dynamic Link Library) for allowing special software configurations. Each network component is designed to interact with the core code through a predefined ensemble of core code requests. A core code request transmitted to a component is a request for participation in the ongoing solution method. The core code has no other access to the component and the component does not need to know what is happening in other component modules.

### B. Enforced programming rules

More powerful languages provide more expressive freedom and power. This is the case of Fortran-95. If there are no guidelines, collaborating programmers can create a cryptic and inefficient code.

In addition to the overall software architecture, several programming rules were enforced in EMTP-F95. These rules allowed to maintain the previously stated balance between readability through high-level programming and efficiency. The gained experience is shared in here by the following list of primary coding practices when using Fortran-95 in EMTP-F95.

- Avoid Fortran-77 constructs and obsolescent features: **`GOTO`**, **`COMMON`**, **`EQUIVALENCE`** …
- Full matrix usage must be avoided unless very small sized matrices are used in some components. The main system matrix is a sparse matrix. All vectors must be allocated automatically by the program. The number of allocations must be minimized by grouping data. Deallocation calls must be minimized and applied mostly for large temporary (nonrecurring) usage arrays.
- Pointer type usage must be restricted to derived types with fields of different lengths. More than one grouping derived types can be chosen to avoid pointers when readability and programming methods are not significantly handicapped. Other pointers must be imitated instead of explicitly declared.
- Several levels of the **`WHERE`** construct can create performance problems when working with large vectors and must be used with caution.
- There are no major requirements for overloading operators for derived types in EMTP-F95. Service routines, however, must hide coding details from users, by providing appropriate overloading interfaces. Such overloading must be available for all possible variations in input arguments. Optional subroutine arguments (new in Fortran-95) can be used to simplify usage of service routines.

On the issue of vectorization, it has been observed that in Fortran-95, vectorization impacts more on readability than speed. The conclusion may be different however, on parallel computers. More experiments are needed also for the indexed parallel array assignment statement **`FORALL`**.

Another important issue is the possible penalty resulting from Fortran-95 data abstraction features. Benchmarking of carefully designed cases and overall software speed indicates that there is almost no penalty on highly optimized compilers (see also similar conclusions in papers found in [4]). Some degradation has been observed when using abstract types with array pointers. This is due to the fact that derived data types cannot contain allocatable arrays directly and pointer type usage is mandatory. The Fortran-200x standard should overcome this limitation by allowing allocatable arrays instead of pointers in derived data types.

### C. Encountered limitations

Fortran-95 does not include intrinsic methods for handling sparse matrices and matrix operations. It is a significant handicap for power system software development. Sparse matrices must be created and maintained using derived types and a substantial programming effort is required to code sparse matrix solvers and to overload functions for operating on sparse matrices.

Fortran-95 is still lacking a large library of intrinsic functions required in typical software engineering. Several array search functions have been added and the support for string manipulations has been dramatically increased, but some basic functions, such as "sort", "unique", "sortunique", "angle" and "interpolate" are still missing. The same comment applies to more advanced linear algebra functions. If Fortran-95 is given a standardized library of

functions similar to MATLAB [2], it will gain unsurpassed capabilities in numerical computations.

Due to graphical user interface issues, the Fotran-95 code must be able to interact with C/C++. Although this is easily available in most compilers, it is not yet part of the Fortran standard. The Fortran-200x standard [6] should provide improved interoperability with the C language.

The fact that using a module automatically provides access to the modules it is using, created complex module dependency structures in EMTP-F95. It is not easy to avoid multiple dependencies. It requires careful documentation of module dependencies to avoid illegal loop chaining of modules.

### D. Performance

Comparing EMTP-F95 performance with the existing Fotrtran-77 based EMTP is not a simple task. It is not only a matter of using different languages, EMTP-F95 used a new set of solution methods and component implementations. In some cases such methods can impact on speed while providing advanced capabilities. EMTP-F95 uses partial pivoting in the solution of its main system of equations and uses a non-symmetric system matrix. Nevertheless, when the coding recommendations stated above are carefully followed, EMTP-95 despite its advanced coding and modernized methods, remains surprisingly as efficient as EMTP for benchmark cases with similar models. This conclusion is applicable for both very large and small system benchmarks.

## IV. CONCLUSIONS

Fortran with its Fortran-95 standard and the upcoming Fortran-200x standard, is advantageously maintaining itself in scientific computation applications.

This paper has demonstrated that Fortran-95 is a powerful and modern language. It is significantly different from its predecessor Fortran-77. Fortran-95 maintains computational efficiency and provides means for enforcing software engineering rules in code robustness and maintainability. This paper indicates that Fortran-95 is not only a logical transition from Fortran-77, it can or should be considered as the preferred language choice for EMTP development.

This paper has presented several Fortran-95 features most suitable for EMTP type software development. It has presented an experiment useful to other developers for similar large scale software development projects.

## V. REFERENCES


[1] Electromagnetic Transients Program (EMTP), Development Coordination Group of EMTP.

[2] MATLAB. The MathWorks Inc.

[3] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith and J. L. Wageneer: Fortran 95 Handbook, Complete ISO/ANSI Reference.

[4] http://www.fortran.com/fortran/metcalf.html

[5] J. Mahseredjian and F. Alvarado: “Creating an Electromagnetic Transients Program in MATLAB: MatEMTP”. IEEE Transactions on Power Delivery. January 1997, Vol. 12, Issue 1, pages 380-388

[6] http://www.j3-fortran.org/