Strategies & Obstacles in Converting a Large Production Application to FORTRAN 90

David J. Gigrich
Structural Analysis Computing
The Boeing Company
david.j.gigrich@boeing.com

Abstract

This report is a documentation of the techniques and problems encountered when converting a finite element program of over 1.4 million lines of software from FORTRAN 77 to FORTRAN 90. The software was originally developed during the late 60's and early 70s in the CDC computing environment by the Boeing Company. This particular application has continued to evolve over the last 25 years and has undergone several Cray ports (Cray 1S, X-MP, Y-MP, & Triton) since 1980. However, it still contained many embedded compiler dependencies (non-ANSI) which greatly reduced its overall portability and inhibited its continued development.

Conversion to FORTRAN 90 is the first in a series of steps toward achieving true application portability. FORTRAN 90 provides the necessary features that replace many older in-house or vendor functionalities, yet are easier to implement and maintain. With adequate planning and preparation, conversion to FORTRAN 90 can be accomplished fairly quickly and efficiently. FORTRAN 90 then provides computing staffs with a cost effective means to achieve application portability.

Table of Contents

1. Introduction -- Rationale for Converting to FORTRAN 90

SGI/Cray predicted the Cray T90 (Triton) CPU follow-on, know as PV+, would be almost twice the speed of their current T90 CPUs. These CPUs were expected to be available in 1999 and many thought they would extend the life of the Triton by several years. However, at the 1998 Cray User Group conference SGI/Cray announced that CMOS computer chips, with ratings as high as one Gigaflop, would be available by the year 2000. Plus, due to the high cost of the PV+ CPUs and their limited marketability, SGI/Cray discontinued all plans to produce them. According to SGI/Cray, they will use cost-effective CMOS technology over the next several years to develop two closely related successor lines, SN2 and SV2. These new product lines are expected to far surpass all current performance levels. Furthermore, the Triton product line is scheduled to be phased out and maintenance support for the product line will be continually decreased over the next few years. In addition SGI/Cray and other vendors have already terminated support for FORTRAN 77.

Termination of support for the FORTRAN 77 compiler required us to look at our legacy systems' machine dependencies. This in combination with determining the next supplier for our high performance computing hardware became the motivational drivers, or business case, for improving the portability of our applications by converting to FORTRAN 90.

2. Advantages of FORTRAN 90

FORTRAN 90 (F90) is currently available for all major hardware platforms ranging from workstations all the way up to high-end supercomputing platforms. Unlike FORTRAN 77 (F77), it is widely supported by most hardware vendors on these types of platforms. The FORTRAN 90 (CF90) compiler was developed by SGI/Cray to support the FORTRAN standards adopted by both the American National Standards Institute (ANSI) and the International Standards Organization (ISO). FORTRAN 90 is a substantial revision to the FORTRAN 77 language standard that provides the analyst with a vastly improved set of commands and structures for developing code. Features of the language found most beneficial are described below.

2.1 Upward Compatible Libraries

The upward compatibility of the libraries and the ability to mix object files provides the luxury of not having to convert all libraries and entire applications at one time. Hence, F77 applications can be loaded with external F90 libraries (data center, Cray, third-party, etc.) or F90 applications can still be loaded with external F77 libraries. Likewise, internal libraries can be loaded in the same manner. This allows for code verification to be performed throughout all phases of the project, especially as each key component is converted to F90. Therefore, the success of the conversion is greatly improved and the risk is reduced.

2.2 Dynamic Memory Allocation

FORTRAN 90 allows for both explicit and implicit Dynamic Memory Allocation. The ALLOCATE feature of the language allows the analyst to explicitly control when memory is made available to an array within a program unit. However, the DEALLOCATE statement must be used to avoid program memory leaks (space becoming unusable) once an array is no longer required. Implicit memory allocation is used for automatic arrays. An automatic array is an explicit-shape array that is not a dummy argument, and which has at least one nonconstant bound. Automatic arrays may be declared in a subroutine or function, but they can not have the SAVE attribute nor can they be initialized. It should be noted that array lengths can now be passed through the calling arguments to implicitly define automatic arrays. These arrays are automatically deallocated once control is returned to the calling routine. The Dynamic Memory Allocation features of F90 have proven beneficial in the phase out of our own cumbersome internal memory management schemes.

2.3 Array Operations (Syntax)

In F90, an array specifier is used to classify an array as explicit-shape, assumed-shape, deferred-shape, or assumed-size. The following F90 declarations illustrate various forms of an array specifier.

```
REAL :: X( 5, 1: 10,  -4 : 3)
DIMENSION Y( 25000 )
! X and Y have explicit shape

REAL :: XX ( m, n )
! XX is an array with an explicit shape and the bounds are not constant

INTEGER :: INFIL ( I : , J ), OUTFIL ( : )
! INFIL and OUTFIL are assumed-shape arrays

REAL, ALLOCATABLE, DIMENSION ( : , : ) :: COORDS
! Declaration of an array with deferred shape

REAL, POINTER :: PTR ( : , : , : )
! Defines a pointer with deferred shape and a rank of three

CHARACTER*20 : :  MYSTRG(20 , * )
! The character string, MYSTRG has an  assumed size
```

FORTRAN 90 provides the ability to perform many array operations based simply on the syntax of the language. An array section is used to denote an array that is a selected portion of another array. There are two subscript forms used to describe a section: vector subscripts and subscript triplets. A vector subscript is an expression that results in a rank-one integer value; the values of the array correspond to the elements of the parent array for a given dimension. A vector subscript allows irregular patterns of elements to be selected. The subscript triplet notation enables a lower bound, upper bound, and a stride to be specified for any dimension of the parent array. In addition, array sections can be used as actual arguments in routine calls.

Many intrinsic procedures have scalar dummy arguments, however, most can also be called with array actual arguments. These are known as elemental intrinsic procedures. Calling an elemental intrinsic function with an array argument, as opposed to a single argument, causes the function to be applied to each element of that array, with each employment yielding a corresponding scalar result value. Hence, the result is an array of the same shape as the argument to the function.

The power of the subscript forms, elemental intrinsic functions, and general array syntax are illustrated in the following examples.

```
INTEGER, DIMENSION (6,5) : : A, X, Y, Z
INTEGER DIMENSION (6) : : B
REAL  DIMENSION (100) : :  R, S
REAL DIMENSION (50) : :  T
X = 0 ;  Y = 0 ;  Z = 0          !  Initialize arrays x, y, and z to zero
X(3, 2:4:1) = 1                  !  Set row 3, columns 2, 3, & 4 to 1
Y(2, 2:6:2) = 2                  |  Set row 2, columns 2, 4, & 6 to 2
Z(1:2, 3:6) = 3                  !  Set row 1 & 2, columns 3 through 6 to 3

A( : ,3) = B( (/ 1,2,4,5,2,6 /) ) !  Set third column of A, where B(2) is used twice

R = SQRT ( S )                   !  Set all element in R to square root of those in S
T  = 1.5 * R(51:100)             !  Multiply last 50 element by 1.5 and store in T
```

2.4 More Intrinsic Procedures

The F90 standard defines intrinsic procedures, which are composed of intrinsic functions and intrinsic subroutines. These procedures may be called from any program unit or subprogram. An intrinsic procedure takes precedence over the intrinsic procedure in a given unit only if the interface is explicit, or it's listed in an EXTERNAL statement in the unit, or it is a statement function. Hence, with the addition of many new intrinsic procedures, the users must use caution to avoid conflicts with their own and supplied library routines. Intrinsic procedures are broken into four classes; 1) Inquiry functions, 2) Elemental functions, 3) Transformational functions, and 4) Subroutines. There are too many new intrinsic procedures to list, but the capabilities they provide can be used to replace many in-house procedures or code segments.

2.5 Derived Data Types & Interface Blocks

FORTRAN 90 allows the creation of new data types, called Derived Data types, which can be constructed from the intrinsic data types and any previously defined new data types. A derived type for manipulating coordinates consisting of three numbers is defined as follows:

```
        TYPE  COORD
                REAL : : X, Y, Z
        END TYPE COORD
        TYPE (COORD) : :  A, B       ! Define variable A & B of type COORD
        A%X = 1.0 ; A%Y = 1.5;  A%Z = 2.2
        B = A
```

Derived Data types can have their individual components handled separately or as a collective.

When referencing an external procedure, it may be necessary for its interface to be made explicit. This can now be done through the provision of an interface block, which is accessible to the scoping unit containing the procedure reference. Interface Blocks allow for defining of generic procedure names, definition of new operators or extension of already defined operators, and they can be used to make explicit interfaces for both external and dummy procedures. The USE statement can be used to make definitions within one module available to another program unit. Hence, modules can share information through USE association.

2.6 Performance Improvements

The Cray FORTRAN 90 compiler has a new optimizer and vectorizer as opposed to what is available in the FORTRAN 77 compiler. It is a more aggressive compiler that will attempt more restructuring. It has the ability to combine loops (Loop Fusion) or split loops into their vector and scalar components to improve overall throughput. It is better at vectorizing large outer loops and can collapse small loops (Loop Linearization) to avoid overhead. In addition, there is more in-lining of code, thus reducing the number of system function calls. Best of all, the F90 compiler is supported; which means problems will be investigated, corrections are timely made, and it will continue to evolve.

2.7 Simplification of Software

The power of the language with its robustness makes it easier to add new features to existing applications or greatly simplifies the development of new applications. This simplification also increases the portability of the software. This leads to reduced development and maintenance cost in both near and long terms, which directly effects our bottom line.

3. Migration Strategies

Our largest Triton application, ATLAS, consists of approximately 1.4 million lines of code, has from 60 to 80 users per month, and is executed around 6000 times per month. The importance of converting this single application alone to FORTRAN 90 and maintaining it for production usage can not be overstated. Time and budget constraints were an issue and the need for good migration strategies to make this project a success was obvious.

Other sites, groups within the company, were solicited to obtain information about the potential problems with such a conversion. The smaller, more modern applications within our own organization were first attempted to build upon this knowledge. Then the following strategies were adopted:

- Verify External F90 Libraries (Cray, Data Center, and Third Party)
- Use FORTRAN 90 compiler to locate noncompliances
  - Triton
  - Workstations (IBM RS6000, HPs)
- Divide and Conquer (subdivide Large Applications)
  - Internal Support Libraries
  - Selective Loading and Testing
  - Precompilers
  - Preprocessors
  - Processors
  - Postprocessors
  - Utilities (e.g., interface codes)
- Verify Small Validation Cases First

The above steps were implemented individually in the order shown to allow for easy isolation of any problems that might occur. The verification of external libraries uncovered implementation differences made by the Data Center, which initially resulted in the failure of our software. Noncompliance compiler issues proved to be too numerous due to the use of Hollerith extensions and had to be ignored. The divide and conquer approach would allow for the conversion and verification of individual sections, application components. Components were classified as performing specific functions or sets of functions; such as libraries, data input preprocessors, processors, and so forth. An application component can range in size from 10,000 to 100,000 lines of code.

This selective building of different sections of the system helped to detect many underlying problems that will be discussed later in this document. It also allowed for a better understanding of validation round-off differences. The ability to intermix the F77 and F90 object files enabled us to minimize the differences seen as each particular section of the application was converted. Even after a section was converted to F90, it was sometimes still advantageous to use the F77 version of it for interfacing with other converted F90 code sections.

4. Problems & Debugging Techniques

There were four types of problems to address during conversion to F90: compilation errors, system aborts, bad results, and numeric differences. Compilation errors and system aborts were the most common and the easiest to resolve. However, bad test results and general numeric differences (round-off) proved equally difficult to correct or explain.

4.1 Compilation Errors

Compilation errors occur because F90 adheres to the language syntax more closely than F77 does. The two most common compilation errors encountered were missing commas in format statements and the referencing of two dimensional arrays (rank 2) as single dimensional arrays (rank 1). The F90 compiler makes it trivial to spot and correct format syntax errors and for that reason they will not be discussed here. However, array syntax errors can be very difficult to correct depending on which one of three methods is used:

- The preferred method is to replace single sub-scripting of arrays of rank 2 with proper double sub-scripting and to also modify the looping structure from single to double.
- Another choice is to convert arrays of rank 2 with this problem to rank 1 (single dimensional) arrays. This usually involves modifying other locations within the code where these arrays are treated as rank 2.
- The last and least preferred method is to simply add, “comma-one” (,1) for the second subscript on these rank 2 arrays. However, compiler optimization improvements, compiler changes, or a different compiler may prevent the over-indexing of columns and different results will be produced.

4.2 System Aborts

It is possible validation cases may fail, not only in the new F90 computing environment, but also under the F77 one due to other changes over time. Therefore, these failed cases should be re-verified under the F77 computing environment. Any such failures that could be duplicated under F77 were considered outside the scope of this conversion project and were directed back to their respective maintenance staffs.

The system trace-back could usually be used to determine a suspect routine for validation aborts due to the F90 conversion. However, the resulting failure of a given routine is usually due to external causes, such as bad results from other routines. The simplicity of component testing, along with the ability to mix F90 and F77 object files allow for quick isolation of the routine that is really at fault. This is done by systematically replacing F90 object files with F77 ones (similar to a binary search) and vise versa until the routine that caused the problem is pin-pointed. Under this scheme, locating one bad routine out of 100 requires no more than seven runs.

The majority of problems (98%) were due to two things, optimization errors and missing routine arguments. Optimization errors can be quickly located by restricting or eliminating optimization through compiler directives in the code or on the command line. Manual scans, tools like cflint, and the interactive debugger (Totalview) work well to determine which calls have missing arguments. Side by side comparison of F90 and F77 my prove necessary when using Totalview. Splitting the code into multiple routines is another good method to pin down the root cause of these types of problems. It should be noted that some system routines (e.g., WRITMS), no longer have optional arguments with default values. The last 2% of the problems were due to arrays being undersized and not becoming a problem until the order of the load changed slightly.

4.3 Bad Results & Round-off Differences

The more difficult part of this project was to resolve why results changed for some of the system validation cases but not others. The same approach, just described (4.2) was used to isolate the code that directly caused the numerical changes. The bad results were attributed to the exact same type problems as above and proved simple to correct once they were located. However, being able to justify new results that were out of tolerance proved to be a major undertaking.

A few validation cases had numeric differences escalate as more and more of the system was converted to F90. This snowball effect would produce vast differences in some of the final results. While this occurred in only a handful of validation cases, it still required the isolating of routines, as well as specific portions of equations within those routines. Isolation of the equations in question and their incoming values verified the round-off suspicions. Extremely small numbers, theoretically zero (1.0E-12 or less), applied to relatively large numbers were always the root of this problem. Results of this nature were flagged for feature reference.

5. Obstacles / Examples / Clean-up

This section will cover a variety of obstacles encountered during the FORTRAN 77 to FORTRAN 90 conversion. Examples will be provided and then followed by several possible solutions. General software clean-up will also be discussed.

5.1 Optimization

The most frequently encountered obstacles were with compiler vectorization and the unrolling of "Do Loops." The aggressiveness of the compiler appears to treat rank 2 arrays as rank 1 for dummy arguments where the second dimension is set to 1. The compiler may also limit the number of trips through a loop based on the declared size of any array. Likewise, arrays that are dimensioned less than their actual intent and are equivalenced to other arrays or common blocks may face the same problem. As an example, a dummy argument dimensioned at 3 columns locally, but defined above with 3 columns for one call and 9 columns in another may have its programmed loop count limited to 3. Since the correct column size is not used in the declared size of the dummy argument, the loop trip count becomes restricted. These types of arrays should use arguments if possible to define the actual dimension sizes or an asterisk if appropriate.

5.2 Hollerith Variables & 32 Bit Loop Registers

Implied looping structures based on IF-GOTO tests may produce undesirable effects when end-loop criteria can contain either numeric and/or Hollerith data. This only makes a difference when the variables containing Hollerith data are zero filled (L format). The reason is that loop registers are 32 bit and a word with less than 5 characters contains all zeros for the right most 32 bits. An implied loop structure established in our software used two exit-loop criteria, the test was for an upper limit on one variable and a keyword on another. However, the compiler generated code for the loop, did not take into account that the values in the array may be text. Hence, checking the loop index against a word containing zero filled Hollerith data caused the loop to terminated prematurely. This problem can be avoided by using only variables with Hollerith data that is blank filled or switching to character data.

5.3 Optional Arguments

The system routine, WRITMS, is used to write data from user memory to random-access file on disk and update the current index. The fifth and sixth arguments to this routine are no longer optional. At one time it was a common practice to exclude one or both of these arguments, as long as the default and desired values were the same. However, with FORTRAN 90 they are now always required. When these arguments are excluded, the results are unpredictable. Depending on the nature of the job, it could abort on the calls to WRITMS, produce bad results, or it may even work. The only viable solution was to correct all WRITMS calls throughout the software.

5.4 Round-off & Common Block Ordering

The aggressive restructuring of the F90 compiler produces code that performs some calculations in a slightly different order than F77, but are still mathematically correct. This results in round-off differences that tend to escalate through each step of a job. Selective testing of individual code components is the best way to isolate these types of problems and to justify final results. Mixing of F77 and F90 binaries can be used to further reduce differences seen due to round-off. External factors, such as new vendor or Data Center libraries also contribute to round-off.

The new compiler may also affect the way in which common blocks are stored in memory. This may result in either job failures or possibly bad results. This occurs because of the software's dependency to require two or more common blocks to be in consecutive memory. Dependencies of this type are often necessary to transfer data quickly from file to memory and vise versa in order to share information between programs. Separating the common blocks causes this sharing mechanism to fail, because the transferred data is either placed in the wrong memory locations or the wrong data is copied to the shared file. The common block order statement can be used in the *segloader* directive file to force the desired order of these types of common blocks. The best solution is to remove the order dependency, but cost and time then become an issue.

5.5 Mixed Arrays

Mixed arrays can be thought of as derived-type objects that may be deferred shape, assumed shape, or assumed size. Mixed arrays contain real, integer, and/or Hollerith data and were used long before the TYPE declaration statement became available in F90. Access to data in these arrays was typically done by one of the three following methods:

> Real and integer arrays of equal size are equivalenced and the appropriate one is used for storing or retrieving data depending on its type. Also the same array could be passed twice through the calling arguments, then treated as two unique arrays (one a real and one integer) to achieve the same effect.
>
> Real values can be moved in or out of integer arrays, across the equal sign, by OR-ing the value to the right of the equal sign with zero (e.g., VAL = INT(I) .OR. 0). The same holds true of moving integer values in or out of real arrays. This treats the item to the right of the equal sign as having no type, thus avoiding a type change.
>
> Internal library functions can also be used to avoid type changes when storing or extracting data from an array containing a mix of data types. The RMOVE function takes a single integer argument and returns a real value. The IMOVE function does the opposite, it takes a single real argument and returns an integer value. While more straight forward, this method had additional overhead due to the function call and prevents vectorization.

The F90 intrinsic function, TRANSFER makes all three of these methods obsolete. It returns a result with a physical representation identical to that of the input source argument, but interpreted with the type of a specified mold. This allows data to be assigned from one type variable to another without a type change being performed. Since TRANSFER is an intrinsic function, it should not inhibit optimization or vectorization. However, it is extremely slow when used with character data, the degradation of one loop resulted in a 100 fold increase to its time requirements. This problem has been reported to Cray and is under investigation. The time increase in this case was avoided by placing the character data TRANSFER statement outside the scope of the loop structure.

5.6 General Clean-up

The FORTRAN 90 conversion was viewed as an opportunity to perform some general clean-up to our software. Where applicable octal formats were replaced with integer types, because memory is now defined in decimal words. All zero-filled Hollerith data has been removed from the system and replaced with either character data or blank-filled Hollerith data. Areas that required revisions to correct problems were improved by replacing loops or routine calls with F90 syntax where practical. Fixed arrays and pointers have been replaced with automatic or allocatable arrays in some instances to improve code maintainability and reduce memory usage.

6. Tools Used

The architecture developed for ATLAS is a modular system of segmented application codes with common executive and data base components. The ATLAS system consists of 50 modules, each performing a well-defined engineering, mathematical or clerical task. Modules range in size from 10,000 to 100,000 lines of code and are managed separately through unique sub-directories. Configuration control of each module is administered through the Source Code Control System (SCCS) via the sccs command. The sccs command is an administrative program that incorporates the set of SCCS commands into the operating system.

6.1 Tools Developed

Initial testing revealed that there were two main causes for compiler optimization problems. These problems stem from declarations for incoming arrays which use a "one" as the last part of their size declarative and Hollerith values that are zero filled (for example, 1L, 2L, 3L, ...). The compiler can restrict loops based on inferred dimensions (ill-defined), rather than actual array sizes as defined above. It also only uses the right most 32 bits for loop testing, hence values that are left adjusted text and/or integer in nature can circumvent loop test criteria. Examples of these problems are illustrated in section 5. An internal F90 program was developed to process the SCCS files, in each ATLAS sub-directory, in order to locate declaration statements that might result in compiler optimization problems. This tool initially identified all DIMENSION, REAL, and INTEGER statements that used a one as the last part of its size declaration. It was soon modified to process line continuations for these declaration statements and to also test for COMPLEX statements of this type. A script was developed around the UNIX grep command to locate variables assigned with Hollerith values that are zero filled. Matching cover procedures were also developed for automating the checking out of the SCCS files. Finally, an internal tool was used to compare previous validation run results against those of the converted system. This tool was used for all phases of the project; it possessed the ability to ignore dates, times, and other such items that were not related to the numerical results. Note that zeros are printed differently in F90 then they are for F77.

6.2 Vendor Tools

The Cflist command used with the CF90 compiler produces formatted listings, tables, and reports about the FORTRAN code. The listing provided loop indicators that identified locations in the code where the compiler had created loop structures not realized by the programmer. The global cross-reference tables and reports helped to identify mismatched calling arguments and undefined variables. The Totalview command was used to invoke the Cray Totalview debugger, which was used for source-level debugging. All these were essential in debugging a routine once it had been identified as the root cause of a problem.

7. System Verification

The policy and philosophy of our programming staff is to verify all software, compiler, and/or operating system changes as thoroughly as reasonability possible. Changes of any type that are not tested, are assumed not to work. The FORTRAN 90 conversion testing was broken down in to three phases; component, validation, and regression testing.

7.1 Component Testing

The system was already logically divided into components (libraries, preprocessors, processors, postprocessors, and utility codes). Each component was separately converted to F90 and then integrated back into the rest of the system (which was F77). This allowed for problems to be isolated to specific sections of code. Test cases from the system library would be selected on the bases of what needed to be tested. There would be anywhere from 7 to 20 test cases run to verify a component of code, depending on the complexity of the code.

7.2 Validation & User Testing

As major portions of this project were completed and verified, then those F90 components were placed in the development (Alpha) version of the system. Hence, our Alpha system contained a mix of both F77 and F90 software throughout the project. A total of 256 validation cases, out of 443 in the library, were executed before allowing all the converted software in to the alpha system. The placing of F90 executables and libraries in to the Alpha system ran over a 11 month period. During this time, the Alpha system was available for both continuous developer and user testing.

7.3 Regression Testing & Release

Once the entire system had been converted to F90, a period of four weeks was required to perform final regression testing. Approximately 40% (190) of the system validation cases were used to verify the correctness and assure the integrity of the system as a whole. This is three times the number usually executed prior to a major block point release. However, with a complete recompilation of the entire system for FORTRAN 90 it was determined to be necessary. The selected validation cases were thought to exercise the majority of the system, with a concentration in the areas most used by the Loads, Flutter, and Weights disciplines. All validation cases met with engineering approval and have been placed on file for future reference. It should be noted that there were other code modifications, in addition to the F90 changes included in the final production release of this system.

8. Resources

ATLAS is a mature system and has many tools and procedures in place for managing the software. The FORTRAN 90 conversion portion of this system release required updates to 2302 routines out of 6423. Which resulted in approximately 6,800 lines of software being modified out of the 1.4 million that make up the entire ATLAS system. The resources required for this project were surprisingly minimal considering the size and complexity of the system. Approximately 370 analyst hours were required from start to finish. Engineering time is estimated to be around 95 hours. This included developing the tools to aid in the conversion process and going through production release. Flow time wise the project ran from March 1998 to February 1999, followed by another month for regression testing and production release. The Cray system is a T916 with 512 MW of memory. Machine resource usage for this project did not affect the company's production usage of the machine.

9. Conclusions

The conversion to FORTRAN 90 on the Cray Triton was one of the most successful migration projects since Boeing's purchase of a Cray 1S in 1980. The converted software executes more efficiently, however, performance differences are still being investigated. Conversion to a newer version of a language, an operating system, or from one platform to another always seems to uncover subtle underlying software problems. However, by continually keeping pace with both language and operating system upgrades, these problems are much easier to manage. Done correctly and in a timely manner, these types of conversions will continue to improve the portability of an application. They constantly force removal of obsolete (deprecated) features, expose hidden software problems, and provide excellent opportunities for clean-up. As an added bonus we are already beginning to see a reduction in cycle time for both sustaining and development activities as a direct result of the additional features provided by the FORTRAN 90 language.

It is much more cost effective to detect and correct errors early. Problems that could have been discovered early, but were not uncovered until the latter stages, prove very difficult to locate and may have downstream repercussions. For example, corrections to a system library routine may change results for several segments of previously verified software. Once corrected, software testing must be repeated to verify that there are no adverse impacts. Thorough testing is necessary at each phase in such conversion projects and it's the only way to guarantee early problem detection.