Technical Report
RAL-TR-2005-008

On the use of "automatic" software quality tools

J.V. Ashby
CCLRC Rutherford Appleton Laboratory
Chilton Didcot
Oxon OX11 0QX
J.V.Ashby@rl.ac.uk

Abstract

Modern computer codes have typically evolved over many years, leading to the dusty-deck problem. We take two large scientific application codes and apply software quality assurance tools to assess whether the use of these tools can help in solving this problem. We find that with care the tools are useful, though far from automatic, and that when used carelessly they can lead to code which no longer works.

1. Introduction

The large and complex codes used for benchmarking computers on “real” applications software represent many years of development time and effort. Often they have evolved through a number of generations and use legacy code. In principle the use of legacy software is a good thing, reducing the effort in developing code and allowing resources to be focused on enhancing the scientific value of the program.

On the other hand, as computers and languages evolve, there is a real risk that legacy codes become out of date and will eventually cease to be usable. With the publication in November 2004 of Fortran2003, there are now five international standards for the Fortran programming language, and several features of the early standards are deprecated or marked for removal.

Academic codes are often developed by many people, although sometimes with one overall architect, who come in, make their contribution to a small corner of the program, receive their PhD or move to the next post-doctoral fellowship and leave without adequately documenting or testing the changes they have made. Many groups are slowly adopting more formal processes of software development such as the use of version control systems such as CVS, formal design methodologies for new projects, etc., but the dusty deck remains a major problem.

There are several tools available to assist with maintaining legacy software and bringing it up to date. The purpose of this document is to assess the usefulness of some of these tools by applying them to two application codes from different scientific areas.

2. The SIC-LMTO and Flite3D codes

The first code chosen for this exercise was the SIC-LMTO electronic structure code written primarily by Temmerman and Szotek [1]. This code solves the Schrödinger equation to find the electronic energy bands in a crystalline structure using the linear muffin tin orbitals (LMTO) approach. Based on Density Functional theory, the code applies a Self-Interaction Correction (SIC) in calculating the exchange and correlation potential. From the comments in the source code, parts of this date back to before 1980, though the main development phase (as documented) seems to have been in the mid 1980's. It has obviously been re-written in part since then as it uses Fortran90 constructs such as modules and dynamic memory allocation. The SIC-LMTO code finally analysed consists of 162 source files containing 166 subprograms with 18246 lines of non-comment source code. In addition there are four modules.

The other code looked at was the Flite3D code from British Aerospace, an unstructured mesh CFD code which solves the Navier-Stokes equations [2]. Initial experience with porting this and with compiling it using the NAG compiler suggested that it would require major restructuring to bring it in line with any of the standard Fortran languages (FORTRAN66, Fortran77, Fortran90/95 or Fortran2003). The solver part of Flite3D consists of 17370 source lines in 88 subprograms and 82 files. There are no modules used in this code. There is a library of ancillary routines used by the code which were not included in this exercise.

3. The Tools

There were two main software tools available for this exercise, Forcheck and Spag. The main tool used was Forcheck from Leiden University [3]. This is a code analyser, which is to say that it parses the source code and checks it for a large number of known potential problems, from lexical errors in the Fortran to unsafe practices. It does no restructuring or prettifying of the source code. The code is checked at several levels, at source line, at the inter-procedural level to make sure that arguments passed between sub-programs are consistent, are not used before they are defined, etc. and finally at the whole program level ensuring that routines are defined once only, are used and not redundant, etc.

Spag, from Polyhedron Software [4], was used occasionally as a back-up tool when Forcheck reported a problem that needed more information. Spag is a code re-structurer – it takes source code and tries to express the implemented algorithm in straight-forward and easily maintainable Fortran. A major problem with this is that after processing by Spag source code is often unrecognisable, even by the original authors. For an example of this, see Appendix A.

The problems reported by Forcheck come in three degrees of severity, Information, Warning and Error. We set the goal of reducing the number of Errors to as close to zero as possible. Ideally we should try to minimise the number of all messages, but clearly identifying Errors and correcting them is of paramount importance.

4. Initial Steps with SIC-LMTO

The first thing that needed to be done was to set up a working environment to use Forcheck. Forcheck has an initialisation file which defines various modes of operation and this is most readily communicated to the program by setting an environment variable. Other environment variables establish the location of the license and other control various options. The variables used are: FCKDIR (installation directory), FCKPWD (location of password file), FCKCNF (configuration file which among other things sets the compiler emulation mode), FCKCPR (controls the printing of a copyright notice) and FCKOPT (a set of default command line options). I simply used the set-up of another user by sourcing a shell script (see appendix B). Such a script should be provided for anyone who wants to use these tools, annotated with how to tailor the script to one's own requirements. Forcheck is capable of emulating many different compilers, the emulation mode used for these experiments was the Lahey-Fujitsu Fortran95 compiler.

We chose to start processing by examining the SIC-LMTO code, and by attempting to reduce the errors as far as possible. With this in mind, the next step was to prune the SIC-LMTO source directories of anything that was not directly relevant. This was done by cross-checking with the makefile and deleting several old versions of routines, ancillary directories and so on. This is a laborious task and although it would be helpful if it could be made automatic, it is hard to see how to make automation foolproof, particularly with the proliferation of file suffixes (.f, .f90,.f95, .F, ...).

Processing by Forcheck was then complicated by the use of both free and fixed form source, and in fact by the muddling of the two, so that, for example, free format code had comments which started with a C in the first column and fixed format code had comments which started with an exclamation mark in column one. To get round this it was decided to transform all fixed form source code to free form using Michael Metcalf's convert program [5], and editing those files already in free form to change any leading C's (also c's and *'s) to !'s. Three files would not go through convert and had to be re-formatted by hand.

It should be noted here that SIC-LMTO makes use of pre-processing using cpp (or fpp where provided) to make specific versions of the program, for example a parallel version using MPI calls to communicate between processes. Forcheck is unable to deal with un-preprocessed code for reasons which Greenough has given at some length [6]. Similarly convert does not deal with un-preprocessed code (even pre-processed code originally had to be edited to remove #-statements left in by cpp, later on use of the -P option obviated this). We therefore used the processed .f files to create .f90 files. For simplicity these were the files edited in what follows, though of course if the use of cpp is to be retained, the edits should be applied to suitably converted .F files.

With this preamble we were ready to let Forcheck loose on the source code.

5. Forcheck processing of SIC-LMTO

The first processing of SIC-LMTO performed using Forcheck failed early in its pass due to the use of modules. When Forcheck came to a USE statement and did not have the information for the module already in its tables, it was unable to continue. This is unsurprising as any (global) variable declared in the module would not be declared in the subprogram that uses it, so Forcheck would be unable to tell whether or not it is being used correctly. The solution to this is to run Forcheck on the module source files alone and build a Forcheck library which is then included in the analysis of the other source files.

At this point running Forcheck on the remaining source code generated 82 Error messages. We worked through these in numerical order (of error message identifier) for want of another approach. It may be that some (numerically) later errors would affect the presence of earlier ones, though we found no clear evidence of this in this case. Some of these errors were trivial due to duplication of files or creation of new, empty files during the initial phase through typographical error, some were potentially serious but turned out to be irrelevant as the code in question was never used by the program, indicating that the makefile needed to be updated as well as the source code. Other errors would make porting the code to a system with a fussy compiler difficult and a few had the potential to do serious damage to the code.

The errors fell into three main categories:

- Syntax errors, deviations from the strict standard:
    - Print statements with no space between print and the format string.
    - Missing repeat counts: in FORMAT statements, the X and H descriptors require a repeat count, leaving it out does not (necessarily) default to 1.
    - Entry points to functions which do not define return values before returning.
- Run-time errors
    - Undefined variables: there were many cases where variables were used before being defined (as far as Forcheck was able to determine). In many cases it may be that an assumption was being made that the variable would be zero, though this is by no means guaranteed, in others it was simply a relic of an old variable that was being used in an unnecessary calculation. In one case a double precision variable was being compared for equality to 1.0D0 (an unsafe practice in itself) as a means of flagging whether a piece of (subsequent) code had been executed. Even in Fortran66 (which this routine clearly was) a logical flag would have been preferable.
    - Variables not allocated. This, and several of the cases where variables were used before

being set, are variables that are used in the MPI version of the code and so the assignment statements involving them probably have no real place in a serial version. To make a definitive statement on this, though, would require expertise in the parallel algorithm being used.

- Subprogram interface errors
  - Inconsistent number of arguments: this led to an error of “variable not defined” when a deep subroutine tried to use a variable that was a missing argument. This variable is actually one member of an array and it is moot whether it would be better to pass the whole array down as the variable to address it is already being passed.
  - Input or Input/output arguments not defined. This is similar to the undefined variables above but where the variable is being referenced having been passed down to a subprogram without having been set.
  - Use of an old trick to do dynamic memory management: before Fortran90 introduced a standard conforming way to do this it was common practice to declare a common block with an array of length 1, switch off the array bound checking and allocate memory from this array. For this to be successful, of course, it was necessary that the common block be right at the end of the data segment (and after the program segment) of the linked executable. Otherwise writing beyond the declared array could damage other variables or even the program itself. Fortunately the use of this in SIC-LMTO is a legacy that has been eliminated through the use of allocatable arrays, so this could be simply deleted.
  - Scalar variables passed to array dummy arguments. There were several instances of this, some innocuous and one certainly dangerous since the first action of the called routine was to zero the three elements of the array it was expecting. There is no way to tell which variables would be in the two memory locations subsequent to the passed variable, or even if they would be the same for two different compilers so potentially anything in the calling routine could have been getting overwritten.

At the end of this phase of processing there were 14 errors remaining in the code:

3x[307 E] variable not defined
4x[312 E] no value assigned to this variable
1x[318 E] not allocated
1x[565 E] number of arguments inconsistent with specification
5x[616 E] input or input/output argument is not defined

These are all errors which may be readily correctable, but which demand an intimate knowledge of the code and its algorithms to understand. In addition there are also 2811 warnings (nearly half of which are about variables not locally defined needing to be SAVEd to retain data) and 1281 informative messages (mostly about unused variables). We went on to reduce these figures to 8 errors, 189 warnings and 270 informative messages, mostly by eliminating unused variables and by using standard intrinsic functions. As noted above, we should ideally try to reduce the number of all warnings and informative messages to zero, since they represent unsafe or sloppy practices.

There are also 30 routines or entry points that are not referenced in the main code. Some of these are due to the use of an old library of file-handling routines, but others represent routines that have been superseded but not eliminated from either the directory or from the makefile. In the case of the file-

handling routines, it would be better to have these as a separate library and include information about them in the Forcheck processing in the same way we included information about modules.

6. Forcheck processing of Flite3D

Following this analysis we went on to consider the Flite3D code. On the face of it this is a simpler proposition with fewer files and routines, although the similar number of lines suggests that the routines are more complex than those in SIC-LMTO. All the code is held in .F files which we pre-process into .f90 files. No pre-processor flags are set, so we do not see, for example, the parallel code or the calls to Vampir routines. The .f90 files are all in fixed (old-style) format. An initial processing with Forcheck gave only 22 errors, 296 warnings and 890 information messages. Using the experience gained with SIC-LMTO, we turned first to the call tree where it appears that there are 6 extraneous roots (routines which are not referenced in the call tree rooted at the main program). Errors not already encountered in SIC-LMTO included: OPENing a file with ACCESS='append' where the ACCESS keyword can only take the values 'sequential' or 'direct', 'append' is a valid value for the POSITION keyword. The bulk of the errors were in passing REALs to routines expecting integers. In most cases this is due to using the old Fortran77 trick of aliasing into an array of length 1 to provide a form of dynamic allocation, though rather than relying on the use of blank COMMON (see above), Flite3D uses Cray pointers to associate a pointer with the array and allocate an appropriate piece of memory. In one case, however, an array is passed which is of different rank as well as of different type, and it is not clear whether this is an intentional piece of aliasing or a programming error.

The use of Cray pointers is common in the code and has been used in several places to provide dynamic memory management which would be better replaced by the use of allocatable arrays. Cray pointers are supported by many compilers as an extension but the declaration statement has a different syntax to the Fortran95 POINTER statement, and there is unlikely to be support for array bound checking.

Flite3D produces many instances of variables not used when processed by Forcheck. While most of them are genuinely unused variables left from previous development phases, a significant number are variables used only in sections of the code not included by the pre-processor. For example the various arrays passed through MPI send and receive routines are declared, but since the MPI flag is not set they appear to be never used. Clearly deleting them from the source would cause problems when subsequently building an MPI version. Ideally such variables should be declared within a conditional compilation block which will only be exercised when needed. Their presence currently complicates the task of reducing the messages about unused variables to zero as it cannot be performed automatically using tools currently available.

The designers of the Flite3D code clearly felt that COMMON blocks were a feature of the Fortran language to be used sparingly. There is some virtue in this view which has led to COMMON being a deprecated feature in recent Fortran standards. However, in this case it has led to subroutines with well over 100 arguments and the associated difficulties of ensuring correct matching of arguments in caller and callee. To an extent tools such as Forcheck, or the use of INTERFACE blocks will help, but the intelligent use of MODULES would be preferable.

7. Corrections which can be made automatically

As can be seen from the previous sections, many of the problems Forcheck highlights in a typical code are multiple instances of the same programming practice. Some of these could be removed by automatic tools, others require human intervention for safety and still others require a deep

understanding of the code to change.

Anything which results in a message at Error level from Forcheck cannot be automatically corrected. An error at this level means that the Forcheck parser has been unable to understand the programmer's intentions, and it is therefore unlikely that any other automatic tool could do so either. This may be slightly restrictive; the lack of a space between PRINT and a format string did not prevent the parser from recognising the statement for what it was, the obvious default to apply to repeat counts for X or H format descriptors is 1, but there is a risk that what appears obvious is not what was meant. Certainly no automatic tool can be sure of correctly initialising variables before they are used. It may be that the code is programmed to assume initialisation to zero, or it may be that a line of code initialising to some other value was accidentally deleted during an edit session.

At the Warnings level, there is more scope for automatic tools to correct indicated problems, and this is even more true at the Information level. The SIC-LMTO and Flite3D codes produced the following Informative messages which could be automatically dealt with as indicated:

- [124 I] statement label unreferenced (remove label)
- [125 I] format statement unreferenced (comment out format statement as it may be needed for debugging purposes or future development)
- [145 I] implicit conversion of scalar to complex (make conversion explicit)
- [250 I] when referencing modules implicit typing is potentially risky (replace with implicit none and force declarations – this should be standard programming practice)
- [347 I] non-optimal explicit type conversion (can be automated using the KIND argument)
- [644 I] none of the entities, imported from the module, is used (remove USE statement, though USE statements should use the USE ONLY form)
- [673 I] not locally referenced, specify SAVE in module to make data global (add SAVE attributes to variables in modules. Most compilers treat variables in modules as SAVEd, but the strict interpretation of the scoping rules allows them to become undefined if the program goes out of scope, just as was the case for COMMON blocks.)
- [675 I] named constant not used (remove)
- [676 I] none of the objects of the common block is used (remove common block and declarations)
- [681 I] not used (remove redundant declarations, but see comments above about conditional compilation)

The following Information messages require programmer intervention

- [284 I] not allocated (a DEALLOCATE without a corresponding ALLOCATE – is this a typo, a missing ALLOCATE or a left over DEALLOCATE?)
- [313 I] possibly no value assigned to this variable (This sort of problem always needs chasing through the logic of the code)
- [315 I] redefined before referenced (There are two assignment statements with a variable on the LHS before it is used in the RHS. Which one is unnecessary?)
- [323 I] variable unreferenced (It is tempting to say just delete the variable, but this is a case of a variable being set and not used, in at least one case because a typographical error slipped in to the

assignment statement.)

- [325 I] input variable unreferenced (Again, this may be a redundant variable or an indication of a typographical error.)
- [341 I] eq. or ineq. comparison of floating point data with integer (Such comparisons should be avoided, but there is no universal best way to do so)
- [342 I] eq. or ineq. comparison of floating point data with zero constant (see above)
- [343 I] implicit conversion of complex to scalar (all of the instances of this are of the form D=C*dconjg(C). The use of COMPLEX*16 is problematic in itself, though supported by many compilers, and it may be that a specific datatype needs to be defined for portability)
- [344 I] implicit conversion of constant (expression) to higher accuracy (This indicated another typographical error, this time in a module. Strong typing would have helped in catching this problem early on.)
- [345 I] implicit conversion to less accurate data type (Not always a disaster, but a programmer should know what conversions are being done and why.)
- [557 I] dummy argument not used (May indicate arguments to subprograms left from testing or earlier versions. If allowed to proliferate can lead to maintenance difficulties.)
- [619 I] conditionally referenced argument is possibly not defined (In many cases this does not matter as the conditional reference is not made until the argument has been defined, but this sort of programming should be avoided if possible.)
- [665 I] eq. or. ineq. comparison of floating point data with constant (see above.)
- [674 I] procedure, program unit, or entry not referenced (Probably a subprogram that has been replaced by another or made redundant in some way.)
- [689 I] data-type length inconsistent with data-type length of function
- [699 I] implicit conversion of real or complex to integer (n=a/b can be made into an integer calculation in several ways with different results)

The Warning messages issued which could be automatically removed were:

- [53 W] tab(s) used (replace with space, in free format it won't matter how many spaces are used)
- [71 W] nonstandard Fortran statement (these are all DOUBLE COMPLEX, use a defined datatype)
- [95 W] nonstandard Fortran syntax (and these are COMPLEX*16, likewise. Also REAL*8)
- [96 W] obsolescent Fortran feature (This covers CHARACTER*n (replace with CHARACTER (LEN=n)), computed GOTOs and arithmetic IFs (these can be removed by Spag) and Statement Functions which can be rewritten as internal subprograms.)
- [98 W] deleted Fortran feature (replace hollerith constants with quoted strings)
- [316 W] not locally defined, specify SAVE in the module to retain data (see [673 I])
- [319 W] not locally allocated, specify SAVE in the module to retain data (see [673 I])
- [413 W] shared DO termination (rewrite do loops to use END DO)

- [438 W] obsolescent terminal statement of DO loop (as above)
- [464 W] missing delimiter in format specification (add a comma following a scale factor edit descriptor)
- [625 W] nonstandard Fortran intrinsic procedure (use of properly derived double complex type should catch most of these)

Warnings that need programmer intervention are:

- [99 W] DATA statement among executable statements (This is strictly allowable Fortran, but offers no advantage and is not recommended.)
- [247 W] assumed length character functions are obsolescent (replace with a subroutine)
- [530 W] possible recursive reference (This was a case of an error reporting routine calling another subroutine which in turn could have called the error reporting routine.)

In many cases, as indicated above, automatic elimination of problems is made difficult by the use in codes of cpp pre-processor commands for conditional compilation.

It is worth going into a little detail over a few of the problems mentioned above as they occur so frequently in programs.

Weak typing, the assignment of any variable whose name started with a letter from I to M as INTEGER and others as REAL unless otherwise declared was one of the features of the early FORTRAN languages. Designed to save programmer time it has probably wasted more in hunting down undefined variables declared implicitly than it has saved. The adoption of IMPLICIT NONE in the Fortran77 standard recognised this and should be part of every programmer's arsenal.

Many programmers are likewise confused by the rules on scoping of global variables. In part this is because compiler writers have taken the easy way out and made global variables static (in the past these were variables in COMMON blocks, nowadays they live in MODULEs). However, according to the standard a global variable comes into scope when the first subprogram which references it is entered and remains in scope until that subprogram exits. If the variable subsequently comes back into scope, there is no requirement for it to contain the same data, or even to be at the same memory location. To make this plain, consider a main program and two subroutines A and B. A is called from the main program, while B is called from both the main program and from A. A and B both USE module C. C becomes active when A is entered, remains in place when A calls B and when B returns to A. When A returns to the main program, C may be destroyed, meaning that when the main program calls B the data in C is no longer there. There are two ways to overcome this: the main program can USE all the modules for the program, though this is not possible if a library writer wishes to use a module, or the module can specify that the data should be SAVEd.

The use of DOUBLE COMPLEX or COMPLEX*16 is non-standard. There are ways of achieving the desired result portably, especially within the Fortran2003 standard using parametrized derived types. Alternatively, many compilers have a module of types which can be used to define such variables through KIND= .

8. Testing and Validation

Having transformed the codes to eliminate the problems and errors noted above the next part of the process should be to run the programs on a comprehensive test deck to make sure that no damage has been done to the code. Each test suite should exercise as much of the program as is practical and come with expected sample output that can be simply compared. In the case of some of the errors noted above, for example the one where memory locations were possibly being overwritten, care would be needed to decide whether any discrepancy in the results was indicative of a bug in the original code or a problem introduced by the transformation. Again ideally the test suite should be run between each change, or at least between small groups if changes so that any effects can be localised.

We had available a set of test problems for each code, which, while not comprehensive, were representative. For SIC-LMTO we had several small datasets and one large one. We initially used a small dataset for Silver which had been used to confirm correct serial and parallel behaviour of the code. Building the program from the transformed source code and running this problem showed up a few minor glitches: some routines had been wrongly identified as unnecessary and deleted, and other routines were duplicated among the source. When these had been ironed out the program ran in serial mode and gave output which was very similar to that before transformation. There were slight differences of formatting and numerical differences at the 9^{th} or 10^{th} decimal place. However, the parallel version of the code failed with a segmentation violation soon after reading in its initial data. The commonest reasons for this are the violation of array bounds and passing incorrect number or types of arguments to subprograms. The latter is covered by Forcheck's inter-unit checking and no such problems were reported. We used the array bound checking flag provided by the compiler to check for the former, and no violations were found. Clearly, though, the process of transforming the code has either corrupted the code in some way or brought to light a corruption that existed originally in the code but was fortuitously not being expressed. With hindsight it is apparent that the test suite should have been run after major edit points in the transformation process so that the change which produced this failure could have been isolated.

In the case of Flite3D we had a test dataset that was suitable for both serial and parallel computation. Both serial and parallel versions gave forces and residuals which agreed with the unprocessed program. The only difference was that the serial version also printed moving body forces which were not produced by either the parallel or the unprocessed version. This can be traced to LMASU, one of the new allocatable arrays which have replaced Cray pointers – it seems as though this is being overwritten or incorrectly passed between program units. Determining where this is happening is complicated by the use of subroutines with very large argument lists, making argument checking difficult. While Forcheck does apply some argument checking, it cannot be foolproof: an automatic tool can check that the call matches the declaration, but not that the appropriate arrays or variables are associated with the arguments by the call.

The unexpected behaviour shown by Flite3D and the failure of SIC-LMTO is an argument for a substantial redesign of both codes. In both cases the structure of the program expresses the underlying algorithm adequately, so the redesign should focus on the data-structures and a clear definition of the scope of variables making good use of modules. Such a redesign requires a good understanding of the semantics of the program, not merely its syntax, so that, for example, all data relating to the mesh could be gathered into one module, and is thus beyond the scope of the present study.

9. Conclusions

This report set out to assess the usefulness of software maintenance and QA tools in improving the quality, reliability and maintainability of typical application codes. The codes chosen were the electronic structure code, SIC-LMTO, the development of which spans over two decades and the unstructured multigrid CFD code, Flite3D, which is documented as being just over a decade old. While there has been some attempt to rewrite and extend these codes in modern Fortran, it is clear that many old constructs remain and the source code has much that is redundant or which has been superseded.

The tool mostly used in this exercise was Forcheck from Leiden University. This is in part due to the lack of availability of other tools, but also because it is a tool which highlights potential problems in a coherent and comprehensive manner, and because by leaving code transformation to the software developer it maintains author recognition.

The use of Forcheck brought many of the problems with the programs to light. Fortunately none of them were serious, at least as far as the tests we were able to run could discover, but they did suggest that the program development process had been somewhat haphazard. This is, sadly, all too typical of academic scientific programs and is partly a feature of the way in which they arise from research groups with high staff turnover and with a focus on producing good science rather than good software. It is to be hoped that judicious use of tools such as Forcheck would assist in producing both.

Most software tools require specific alterations to a user's environment to operate successfully. The provision of well-documented and well designed sample scripts can make these alterations painless.

It would be nice if the only action necessary from a user was to type forcheck *.f90 (for example) and the code was magically transformed into an error-free state. As we have seen above, the interpretation and understanding of errors and how to resolve them still requires a good deal of human input, some of it from an expert in the code and the science behind it. For this reason we were unable to reduce the number of errors to zero, though we did manage a reduction of over 80%.

A major consideration in designing a software development process which incorporates quality conformance testing of the sort discussed here is the inability of most tools to deal with pre-processor files (*.F, referred to as ur-source by analogy with ur-text). We ignored this problem by simply working on a pre-processed version of the code and making changes to that. Clearly the errors still exist in the ur-source, and will re-surface if this is pre-processed (for example when creating an MPI version of the program). Making the changes to the ur-source and re-running the pre-processor would help in this problem, and any process which attempted to be comprehensive should cover all relevant combinations of pre-processor flags. This is in itself an excellent reason for keeping the use of pre-processors to a minimum.

Finally, any changes to a program should be validated against a comprehensive execution test suite with suitable scripts to run tests and expected output. Changes which add functionality to a program should generate additions to this suite.

Appendix A Source Transformation and Source Recognition

To illustrate the problem of author recognition that may arise when source code is automatically transformed by software tools such as Spag, we have used Spag on the simple shell sort routine used in SIC-LMTO.

First we show the original source code:

```
       subroutine ishell(m,n,iarray)
!- shell sort of a array of integer vectors

!
-----------------------------------------------------------------------
-
!i Inputs:

!i   m     :number of components in iarray

!i   n     :number of elements in iarray

!i   iarray:array to be sorted

!o Outputs:

!o   iarray:array to be sorted

!
-----------------------------------------------------------------------
-
      implicit none
! Passed parameters:

      integer m,n,iarray(m,0:n-1)
! Local parameters:

      integer lognb2,i,j,k,l,n2,nn,it,mm,mmm
! Intrinsic functions

      intrinsic float,int,log

      lognb2 = int(log(float(n+1))*1.4426950d0)
      n2 = n
      do nn = 1, lognb2
        n2 = n2/2
        k = n - n2
        do  11  j = 1, k
          i = j - 1
    3     continue
          l = i + n2
          do  15  mm = 1, m
            if (iarray(mm,l) - iarray(mm,i)) 16,15,11
   16       continue
            do mmm = 1, m
              it = iarray(mmm,i)
              iarray(mmm,i) = iarray(mmm,l)
```

```
              iarray(mmm,l) = it
            enddo
            i = i - n2
            if (i .ge. 0) goto 3
            goto 11
   15     continue
   11   continue
      enddo

      END
```

After Spag this is transformed to:

```
!*==ishell.spg  processed by SPAG 6.50Rc at 16:49 on 20 Oct 2004
      SUBROUTINE ISHELL(M,N,Iarray)
!- shell sort of a array of integer vectors
!
----------------------------------------------------------------------
-
!i Inputs:
!i   m     :number of components in iarray
!i   n     :number of elements in iarray
!i   iarray:array to be sorted
!o Outputs:
!o   iarray:array to be sorted
!
----------------------------------------------------------------------
-
      IMPLICIT NONE
!*--ISHELL13
! Passed parameters:
      INTEGER M , N , Iarray(M,0:N-1)
! Local parameters:
      INTEGER lognb2 , i , j , k , l , n2 , nn , it , mm , mmm
! Intrinsic functions
      INTRINSIC FLOAT , INT , LOG

      lognb2 = INT(LOG(FLOAT(N+1))*1.4426950D0)
      n2 = N
      DO nn = 1 , lognb2
         n2 = n2/2
         k = N – n2
         DO j = 1 , k
            i = j – 1
 20         l = i + n2
            DO mm = 1 , M
               IF ( Iarray(mm,l).LT.Iarray(mm,i) ) THEN
                  DO mmm = 1 , M
                     it = Iarray(mmm,i)
```

```
                     Iarray(mmm,i) = Iarray(mmm,l)
                     Iarray(mmm,l) = it
                  ENDDO
                  i = i – n2
                  IF ( i.LT.0 ) GOTO 50
                  GOTO 20
               ELSEIF ( Iarray(mm,l).NE.Iarray(mm,i) ) THEN
                  GOTO 50
               ENDIF
            ENDDO
50       ENDDO
      ENDDO

      END
```

Factors which may affect the author recognition, aside from the good aspects such as replacing the arithmetic IF statement by an IF-ELSEIF construct include the renumbering of statement labels and the capitalisation or uppercasing of arrays and keywords. Most of this is configurable in Spag, and care needs to be taken in choosing which options to allow.

Appendix B Script for setting Forcheck environment

The following script is appropriate for users of the C-shell and its derivatives. This script should be 'source'd to ensure that the variables it defines become part of the current environment. If it is executed then a fresh shell will be established, the variables created and the shell destroyed as the script exits.

```
# Install directory for Forcheck
setenv FCKDIR /home/mathsoft/FORCHECK
# Forcheck license password file
setenv FCKPWD /home/mathsoft/FORCHECK/fckpwd.pwd
# Compiloer emulation configuration file
setenv FCKCNF ${FCKDIR}/laheyf95.cnf
# Don't print copyright notice
setenv FCKCPR QUIET
# Default options – in this case use Fortran95 standard and free format
setenv FCKOPT "-f95 -ff"
# Add to PATH so forcheck can be invoked simply
setenv PATH ${PATH}:${FCKDIR}
```

Users of the Bourne shell and its derivatives should source the following script:

```
# Install directory for Forcheck
FCKDIR=/home/mathsoft/FORCHECK
# Forcheck license password file
FCKPWD=/home/mathsoft/FORCHECK/fckpwd.pwd
# Compiloer emulation configuration file
FCKCNF=${FCKDIR}/laheyf95.cnf
# Don't print copyright notice
```

```
FCKCPR=QUIET
# Default options – in this case use Fortran95 standard and free format
FCKOPT="-f95 -ff"
export FCKDIR
export FCKPWD
export FCKCNF
export FCKCPR
export FCKOPT
# Add to PATH so forcheck can be invoked simply
PATH=${PATH}:${FCKDIR}
```

References

1. W.M. Temmerman, A.Svane, Z. Szotek, H. Winter and S. Beiden, “On the implementation of the Self-Interaction Corrected Local Spin Density Approximation for d- and f-electron systems”, Lecture Notes in Physics **535**: “Electronic structure and Physical Properties of Solids: The uses of the LMTO method”, ed. H. Dreyssé, Springer-Verlag, ISBN 3-540-67238-9 (2000) 286-312.
2. Parallelisation of FLITE3D, D. R. Emerson and M. Ashworth http://www.cse.clrc.ac.uk/ceg/flite3d/flite3d.shtml
3. Forcheck, a Fortran analyser and programming aid, http://www.forcheck.nl
4. http://www.polyhedron.co.uk/
5. Metcalf M. convert.f90 available from ftp://ftp.numerical.rl.ac.uk/pub/MandR/
6. Greenough C. “The Transformation of Legacy Software: Some Tools and a Process.” RAL Technical Report TR-2003 012 (2004)