A Test Harness TH for Numerical Applications and Libraries

Brian T. Smith
Numerica 21 Incorporated
Angel Fire, New Mexico, USA, carbess@swcp.com,

Abstract. TH is a test harness to facilitate the development of scientific software. The operational model is the comparison of the results from running two versions of an application code to ensure the results are equivalent. First, TH is installed into an existing application code that runs to completion on a set of data. Installation tools provide a readily-modified default initial configuration. The application code with TH installed is run in generate mode to create a monitored data file. A second version of the application with TH installed is run in check mode, comparing the current results with the original results. Features include specifiable criterion for data comparison, and a design that facilitates the installation of TH into codes written in any programming language and in parallel SPMD codes. Once installed, TH can be deactivated, permitting the same code to be maintained with and without the test harness in use.

1 Introduction

TH is a test harness to facilitate the development of scientific software, currently in Fortran. The test harness is based on the operational principle that the coder wants to ensure the software is producing the "same" results before and after some changes were made to the software, or between the software running on two different platforms. The test harness is used by taking an existing scientific application code that runs to completion on a set of data, inserting "include" lines by hand or by script for large application codes. The instrumented application code, with an input file specifying the variables that are to be monitored, is analyzed by a software tool called the `builder`. The builder tool creates from provided template files the application-specific "include" files needed to run the application software with the test harness installed. The application code with the test harness installed is then run in generate mode to create a data file against which modified versions of the software or versions on different platforms are run to detect any significant changes in the results.

The various runs of the software might use different versions of the application code with the test harness installed into them in the same way. Typically scientific applications may be enhanced to improve efficiency, to improve capability, to verify the correct porting to a different platform, to modernize the code, or to check the results with different compiler options, typically optimization flags. The test harness compares current values of data variables with previously obtained values and reports only those that are "significantly" different. The coder only specifies the variables whose data values are to be recorded and compared, and the criterion and tolerances for the comparison; writing and reading the past values and all comparisons are implemented by the test harness, and the test harness reports significant differences that violate the comparison criteria.

In contrast, comparisons by hand of results before and after comparable runs are tedious, error-prone, very difficult, or often impractical because of the different impacts of rounding errors. The test harness addresses this issue by providing data comparators that under programmer specifications check for near-identity rather than identity, comparing results based on relative or absolute tolerances, or both. Comparisons for arrays are facilitated by array comparator routines provided by the test harness. When differences are detected, the diagnostic information printed indicates what the tolerances should be to pass the checking procedures and the first element in array element order that is significantly different. In addition, as needed, the comparators can be made to ignore any differences.

The paper describes the test harness, its operation, its testing, and experience with it in testing various application codes. In section 2, more detail is provided on how the test harness is to be used; in section 3, the tools as well as the input to these tools that build the test harness into the application are described. Section 4.0 describes the tests that have been used to evaluate the test harness; Section 5.0 describes the plans for future enhancements to the test harness and Section 6.0 provides a summary and conclusion.

2.0 The Problem and the Test Harness Concept

A problem encountered by many code developers and code maintainers is to determine whether a large code continues to operate correctly or in the same way as it did in the past after modifying or enhancing it. The modifications or enhancements might be, for example, to improve the code's efficiency, add new features, compile it with different optimization flags, or make modifications that permit it to run on a different architecture. In such scenarios, the code developer has a collection of test to rerun and wants to ensure that the code behaves the same way where the changes should have no effects on the numerical results.

The test harness is a tool to aid in this testing process. Often even modest changes such as rearranging computations or performing them in a different order will change the results numerically. Comparing numerical results by hand is a tedious and error-prone process in which in many cases all the results are different but only by a amount consist with the stability of the numerical computation. The test harness is a tool that permits the comparison of two sets of results using error tolerances specified by the user. The current version of the test harness permits the comparisons

of pairs of variables (scalars, arrays, or structured objects), one from a version of the application code run in "generate" mode and the corresponding variables from a version of the code run in "check" mode. The comparisons to be performed are specified by giving the variable by name, the subprogram or program the variable is in, the location of the comparison (for example, on entry to a procedure, exit from a procedure, or at any arbitrary point in a procedure), and the tolerance used for the comparison.

Control of the comparisons and what is compared is specified by input to the harness tool `builder`. The tool creates modifiable application-specific "include" files and a module with module procedures containing the comparator code. The comparator code is in a separate module that can be readily modified by the coder to handle special cases but the typical ordinary cases are provided by default in this separate module. In this way, unusual comparisons, say for combinations of specific elements of arrays or non-intrinsic derived types can be coded into this separate module and references to them can be placed in specific "include" files as needed.

2.1 The Approach in Detail

Figure 1 illustrates the modes of operation of the application code with the test harness installed. The figure assumes the test harness has been installed in two versions of the application code. The flow on the left shows the application code

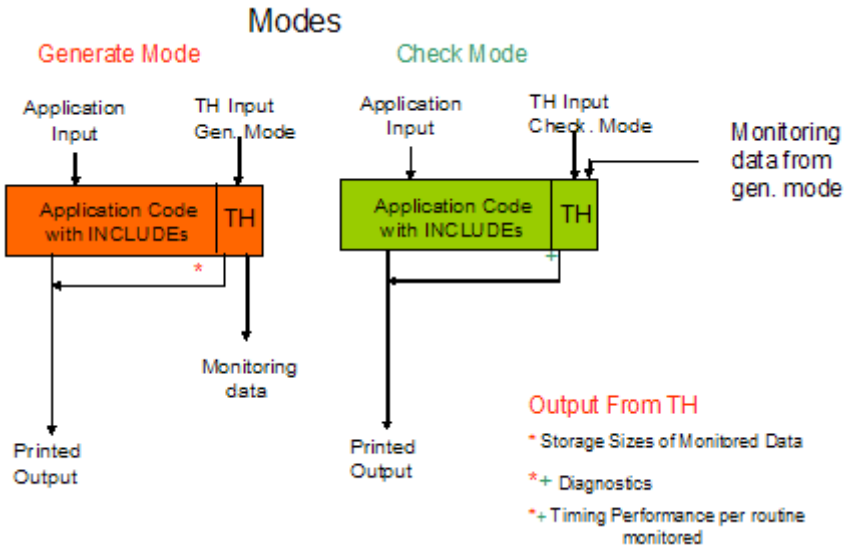

Figure 1: Operation of the Test Harness

with the test harness installed into it where the test harness is run in "generate" mode. The application code reads its input and performs all of its computation as usual, producing its usual output. The test harness records the values of the variables

that it is monitoring in an unformatted sequential file. The recorded data includes both application code values and data from the test harness, representing the execution sequence of the monitored procedures. The test harness performs certain consistency tests, providing diagnostics when it detects errors. In addition, it accumulates and prints the frequency counts, execution times, and data sizes written by each monitored application procedure (these are indicated in Figure 1 with an asterisk on the printed output line from the test harness). This latter computation and output can be turned off by input to the test harness.

The flow on the right shows the application code run in check mode. The diagram is essentially the same except that the test harness reads the monitored data generated from the unformatted file created from the test harness code run in generate mode. However, the test harness runs in a different way; this time, it compares the values of the monitored variables with the data from the unformatted file. The comparisons are performed as specified to the builder tool and any variance from the specified tolerances are analyzed and diagnosed on a separate output error unit, specified to the test harness. The analysis process prints the value that the tolerances must be to pass the comparison tests and recommends whether the comparison should use a relative or absolute tolerance criterion. In addition, the test harness again generates frequency counts and execution times, providing an indication of the cost of the monitoring and evaluation, particularly when compared with the similar data generated by the test harness in generate mode (this output is indicated by a plus on the printed output line from the test harness icon in Figure 1).

The operation of the application with the test harness installed implies a relationship between the flow in generate and check mode. This relationship is that the two runs (likely with different versions of the application) must visit the same monitored procedures in the same order. Consequently, the monitoring must be with variables and procedures that are not expected to be vastly different in structure. This is often the case for changes that involve differences of optimization, code modification for efficiency improvements, and a whole host of other practical reasons why the codes are different.

To detect only differences in code flow, the test harness can installed in both versions so that it monitors no application data but monitors only code flow through the monitored procedures; this is sometimes very useful to know in diagnosing the cause of different results in two codes that are supposed to produce the same results.

Figure 2 shows the kinds of “include” lines added to a main program; these “include” lines are required in the main program but are only needed in procedures whose data or operation is being monitored. The “include” lines are similar for any subprogram unit selected for monitoring. An “include” line is required for each additional probe point and each exit from a procedure being monitored. No application-code data need be monitored, in which case the TH is recording and/or comparing an execution trace of the application code through the monitored application subprograms.

Currently, the “include” lines are inserted by hand. The particular “include” lines depend on whether the procedure is an internal procedure, module procedure, and external procedure. Also, the “include” line insertions are different for F compliant code [1] versus Fortran 90/95/2003 [2] compliant code. The “include” line insertions are specified in detail in the User’s Guide [3] for the test harness; a future tool,

currently designed and partially implemented, will perform the required insertions in all procedures of an application and will monitor all eligible variables that are potentially referenced by the procedure and potentially defined by the procedure at entry points to and exit points from the procedure. As shown in Figure 2, the "include" lines are spelled with the keyword "binclude", in order to distinguish them from Fortran include lines; whether they are spelled "binclude" or "include" or some other way is specified in the input to the builder tool.

Once the procedures for monitoring are selected, the variables in these procedures are determined. The "include" lines are inserted, and the builder tool builds the application specific "include" files. Then an includer tool is run in one of two ways; one way builds the application with active version of the "include" files illustrated in Figure 1; a second way builds the application with the test harness disabled or inactive, thus providing a production version of the application code where there is no interference from the test harness. These alternative ways of generating executable code permit one version of the code to be maintained and at application build time, the user has the choice of building the application for testing or production.

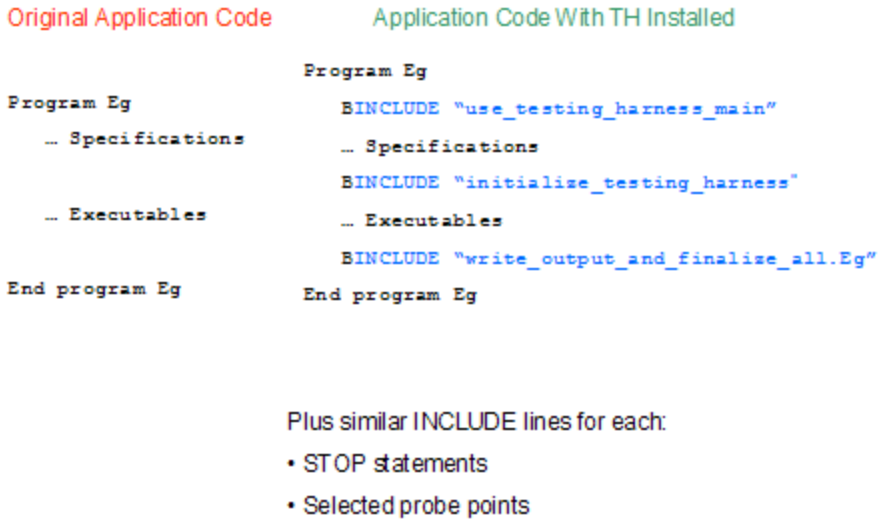

Figure 2: Installing the Test Harness in the main program - an example

2.2 The Usage Scenario

Figure 3 illustrates the use of the test harness over time, with time progressing from the left side of the figure to the right side; also the code is being developed and enhanced as time progresses. The figure actually depicts the development of the builder tool itself. Initially, an earlier version of the application code has the test harness installed into it; this is depicted at the left of the figure. A test suite is obtained or developed for this early version and the application code with the test

installed and activated is run in generate mode for all the test cases. With the test harness in the application code, the application code is modified, being improved in efficiency or capability, or just ported to a different machine. The code is then run in check mode with the same test cases and the new results are compared with the previous results. When corrections are made, the code is rerun and the results compared again until the results are acceptable.

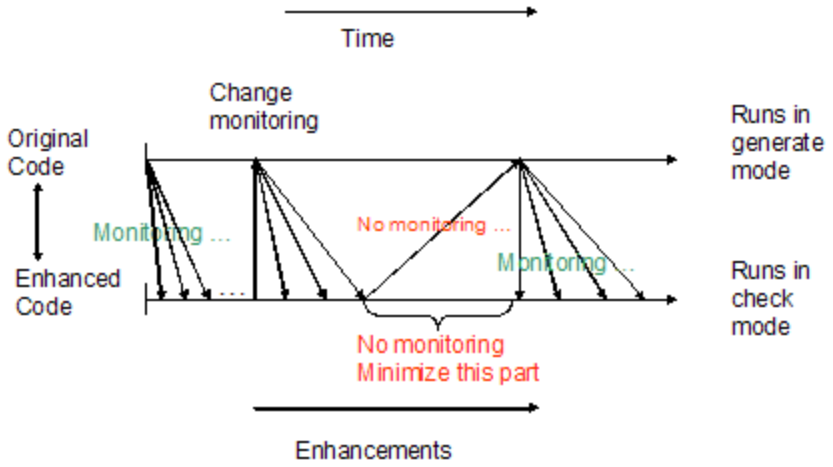

Figure 3: Test Harness Usage Scenario

At some point, what is monitored may be changed (as indicated in the left middle of the figure) and maybe new test cases are developed; at such a time, the application with the test harness installed is rerun in generate mode, creating a new collection of recorded results, which are then compared to evaluate the effect of the further changes and enhancements. Occasionally, changes will be made that cannot be monitored with the test harness (for example, changes that cause monitored execution flow to be different between the code run in generate mode and check mode) unless what is monitored is changed; this is depicted in the figure just left of the middle of the figure. Finally, the code reaches a state where its performance is satisfactory (extreme right of the figure) and it is put into production. The test harness can remain installed in the code but made inactive ready for future activation and further evaluation when investigating new test cases.

The above scenario was followed in the development of the builder tool itself. An early version of the test harness was installed in the initial version of the builder tool code. The builder tool uses floating point computations sparingly but the test harness allows comparisons of data values of all intrinsic types; comparisons for identity of integer, logical, and character values immediately indicated unexpected in parts of the code that used to work before certain enhancements were made.

Once the test harness is installed in an application, it measures and displays the execution times and counts of the monitored subprograms, and the sizes of the check data created by each monitoring probe. This information is useful in controlling the

size of the check data files. This becomes a problem either because the place where the data is monitored is executed too frequently or the data monitored is too voluminous. Although the test harness has been successfully used to monitor data where the check data file became as large as 10 gigabytes, the monitoring process may be timing consuming and may need to be curtailed.

To curtail the size of the monitored data, the frequency that each probe generates or check its data may be decreased. Also, for arrays, the elements that are monitored may be specified so that an entire array is not monitored. For each of these changes, the test harness has to be reinstalled into the application using the builder tool to implement the changes.

In addition, the comparison routines can be replaced by comparators that compare norms of matrices or compute condition numbers and compare them, or evaluate the differences in the result in any desired way. Such replacement comparators may be motivated to reduce the monitored data sizes but may also be motivated to focus on the comparisons of significant aspects of the computation that are particularly meaningful to the application. In each of these cases, the test harness has to be rebuilt into the application code by rerunning the builder tool.

3.0 Installation of the Test Harness in an Application Code

The installation of the test harness into an application code requires three steps. First, the "include" lines are inserted into the application code as per the instructions given in [3] and exemplified in Figure 2. Typically, three or four "include" lines are inserted per procedure monitored; the text of the "include" files USE statements, CALL statements, testing harness initialization code, and test harness cleanup code. In addition, an "include" line is needed for each probe that is additional to those at each entry and exit point from each procedure monitored. Any number of additional probes can be inserted at any desired points.

The second step is to prepare the input to the builder tool. It is typically one input file (the builder can be run on parts of the application code but each run of the builder tool requires an input file). This input file specifies the following:

- The input file containing the application code with inserted "include" lines.
- The output file containing the application code modified appropriately to use the test harness.
- The input directory containing the template "include" files
- The output directory containing the specific "include" files activating the test harness.
- The output directory containing the specific "include" files for an inactive test harness.
- The "include" line keyword used in addition to BINCLUDE processed by the builder tool; if it is spelled "include", it will process Fortran INCLUDE lines and if blank or spelled some other way, it ignores the contents of the Fortran INCLUDE files.
- "code" blocks specifying
    - the name of the procedure to monitor
    - the variables monitored on input to the procedure

- the variables monitored on exit from the procedure
- the variables monitored at each specific named probe point

For each variable monitored, the relative error tolerance (relative to machine epsilon) and the absolute error tolerances can be specified; if not specified, the tolerances are zero, requiring identity to pass the comparison tests. There are a collection of other properties that can be specified, such as lower bounds for arrays and selected ranges of elements of arrays to check (required for assumed-size arrays).

The third step is to run first the builder tool with its input and then the includer tool with its modest input (location of directories, input file, output file, and logical unit specifications to avoid portability problems with the tools specifying input/output units not supported in the same way the default units are assumed to be used). The result of running both tools is either the application code with an activated test harness installed in it or an inactivated test harness. In the first case, the application can then be run using the scenario exemplified by Figures1 and 3. Code enhancements should be implemented in the version of the code after the first step above so that the enhancements can be installed without repeating the building steps unnecessarily.

3.1 Code Insertion into the Application Code

The code lines inserted by hand are “include” lines. However, there is a minor difficulty in this process when the insertion point is prior to any labeled statement, like a RETURN, END, or STOP. In addition, there is no place to insert a probe after an IF statement test but before the object of the IF statement is executed. The problem is that the insertions are essentially INCLUDE lines referring to files containing executable code and must be on a separate line, as required by the Fortran language. To make code insertions in these case, the code has to be restructured using block IF constructs or labeled CONTINUE statements.

As mentioned in 5.0, an additional tool is under development that will insert the required lines in specified places, avoiding such changes by hand. This tool is rather complex because it also is creating default variables to monitor. Refer to 5.0 for a complete description of this and other tools being planned.

In all cases, the source form of the inserted code satisfies both the requirements of fixed form Fortran source or free form Fortran source. Thus, the application code can be written in either free or fixed source form and in Fortran 77, Fortran 90, Fortran 95, Fortran 2003, or F compliant code. The compiler used to compile the application code with the test harness installed must be either a Fortran 90/95/2003 or F compliant compiler.

3.2 Building the Test Harness from Templates

The builder tool builds the test harness into the application by processing template “include”. Typically, the builder creates a large number of such specific include files and places them in a separate directory, called the active directory. A

second directory of "include" files is created by this tool, which is referred to as the empty or inactive "include" files. A second tool, called the includer, is then run which builds the complete application from the directories of files created by the builder tool. The input to the includer tool specifies which directory of specific "include" files are to be used, building the application using either the active test harness or the inactive test harness.

The template files are in detail in [4].

3.3 Test Harness Input

The test harness input is read from standard input by default and specifies the following items:

- The test harness mode; the mode is either generate, check_and_continue, or check_but_terminate. The mode check_and_continue continues to the completion of the application code no matter what violations of the comparison tolerances are detected. The mode check_but_terminate terminates after the completion of the checking for a probe when the first difference violating the comparison tolerances is found.
- Print performance execution times or not. The execution time for each monitored procedure is printed.
- Print storage information in generate mode. This information includes the size of the data in the unformatted file generated per procedure; the unit size is processor-dependent but on most systems is either a byte or single precision or default real word.

A named input file can be provided to override the input/output units used by the test harness and avoid conflicts with those units used by the application. If not provided, default units are selected. These units specify the test harness input described above, the unit for debugging output, the unit for the unformatted check data file, and error diagnostic unit.

3.4 Performance Of the Test Harness Tools and the Harness Itself

The builder and includer tools perform their builds very quickly, typically in considerably less time than it takes to compile the code, with or without the test harness installed.

The test harness itself does impact the performance of the application code but that depends very directly on the amount of data monitored and the speed of the input/output system. For modest data sizes, say less than a few megabytes, the performance penalty is at most a few seconds; in generate mode, it is hardly noticeable; in check mode, it is typically less than 3 seconds. It is more in check mode because reading unformatted files takes longer and the comparison checks, especially for large arrays, can take some time. As another measure of performance, the test harness added approximately 30 seconds to an application run when creating a 10 gigabyte test file while the application code ran for approximately 10 minutes with and without the test harness code activated.

As suggested above, performance is very dependent on how much data is being monitored. To see how much data is being generated and how long it is taking, the test harness should be run, specifying in its input, “performance” and “storage” in generate mode and “performance” in check mode. Analyzing the tables of storage sizes and performance per procedure will provide an understanding of where the large data and costly performance is coming from. As mentioned above, decreasing the recording and checking frequency, decreasing data monitored per procedure, or decreasing the number of procedures monitored can address performance issues, if and when, costly performance issues are encountered.

Performance timing is always an issue and is very system dependent. The test harness code uses one of three alternative timers; a Fortran 90 timer, and two alternative timers that are available on many Unix/Linux systems. The timing performance procedures are in a visible separate module and are selected by commenting out Fortran lines. In addition, your own reliable timer can be substituted for any of them, provided the module procedure returns time in units of seconds. The Fortran intrinsic procedure CPU_TIME is used by default.

4.0 Experiences with Testing the Test Harness

The tests performed to date have been of varying sorts. First, five serial codes taken from a tutorial for PARAWISE from Parallel Software Products Inc [2]. The tutorial involved the use of test codes implementing a simple 1-D Jacobi algorithm, a 2-D heat conduction and diffusion problem, a 2-D steady state flow prediction code, a simple unstructured mesh algorithm, and the serial version of the NAS-PAR benchmark APPLU. All five application codes were written Fortran 77 using fixed source forms. Each code was then rewritten completely using Fortran 90/95 constructs to the fullest extent, and in particular arrays and modules, including the conversion of block data subprograms to modules. In addition, the source code of the Fortran 90/95 versions is written in free source form. The tests involved running the original Fortran 77 code in generate mode and the Fortran 90/95 code in check mode, comparing the results. In all cases, after fixing various errors in the initial conversion process, the two versions create consistent or comparable results as measured by the test harness.

In two of these codes, the results were most interesting. In the NAS-PAR benchmark, the initial results showed a drastic difference after fixing the conversion errors detected by the test harness. The cause of the final differences was the use of intrinsic function SUM to perform sums rather than use of loops to perform the summations. After several runs using the test harness, the cause of the difference was quickly rectified by using the loop form of the computation rather than the SUM intrinsic in the sense that the differences were reduced to small numbers of rounding errors.

Using the test harness with the 2-D steady state flow prediction code represented a second interesting test case. The input data set used, if run for a large enough number of iterations, shows numerical instability. The original Fortran 77 and the converted Fortran 90 versions display the instability at different numbers of iterations. Using the test harness, one is able to monitor the instability and see how

the two versions exhibit the instability at different points in the run. The differences are again attributed to the use of intrinsic functions rather than code loops and can be exposed in the comparative runs using the test harness.

These five test cases are interesting from another aspect. Although the Fortran 90 codes are drastically different than the original Fortran 77 codes, the computation order remained the same using the same input test file. This was partially accomplished by not monitoring data in places and procedures where the execution sequences were different between the two versions of the code. But it also illustrates that major changes in code structure can be performed without distributing the execution sequences in a way that matters to the monitoring.

Initially, a parallel SPMD code was used in developing the design of the test harness. Quickly, it became apparent that a solid serial version was needed first and a version for parallel SPMD code would readily follow. However, the parallel code test harness code has not been tested with the latest version of the test harness, desiring to complete the various needed tools for serial code before addressing in detail the parallel code issues.

4.1 Installation Tests

The test harness is distributed with five installation tests. The simplest one is manufactured test code (that is, constructed to test a few difficult corners of the testing process) but it otherwise is performing silly computations. In particular, the test uses external procedures with entry procedures, both functions and subroutines, including recursive procedures. It is written in two forms, fixed and free source form.

A second set of manufactured installation tests is a collection of 32 programs, one of which is the above test. These additional tests include internal procedures, module procedures, modules, internal and entry procedures in module procedures, and recursive procedures, both functions and subroutines. Also, several of the tests use test code that is F compliant where the “include” lines, to remain F compliant, are required to be different.

Several of these 32 tests check a rather nasty implementation issue with the test harness. The test harness operates by creating internal procedures within the application procedures that perform the generation of the check data files and comparison of the check data files with data from the application when in check mode. This approach is used so that the scope of application procedure is inherited by the harness procedure and thus has access to application data environment without the use of common blocks or long lists of arguments. But because Fortran does not permit internal procedures within internals procedures, this approach had to be replaced by generating code in line to create and read the check data files. In both cases but particularly in the latter case, there is the possibility that variables created for use exclusively by the test harness clash with (have the same names as) application variables. To avoid this conflict and consequent limitation, the generated variable names may be prefixed by a user-specified letter string that is specified by the user to avoid name conflicts. Several of the test cases test this capability.

As a final point, these manufactured test cases can be used as tutorial examples; they illustrate the locations and forms of the inserted “include” lines that are inserted

into the codes depending on the procedures and modules, including entry procedures, used in the application code.

4.2 Installation Application Tests

Three of the five application codes are also provided as installation tests. They include the simple 1-D Jacobian code which consists only of a main program and can again be used as an example to follow and study. The other two application installation tests are the APPLU NAS-PAR code and the 2-D steady state flow prediction code that shows the instability with the test input file provided. It is valuable and informative to look at how the tolerances where specified and had to be relaxed so that the test harness would allow the Fortran 90 version to complete execution.

4.3 Documentation

The documentation consists of an overview description [4] of the test harness and a User's Guide [3]. This overview describes the motivation and objectives of the test harness tool, its input, its default output, and its debugging output. It describes the tests in detail that were used to develop the test harness and the role and purpose of each of the "include" files inserted into the application code. Finally, it lists the planned enhancements to the test harness for the next year or so.

The User's Guide gives detailed instructions on where and what include lines need to be inserted into the application code. The User Guide also describes the builder tool's input, which specifies the procedures to be monitored, the variables in those procedures to be monitored, and the particular probes monitoring those variables. Along with the variables are specified their type, kind, rank, and dimension information, how much of an array is to be monitored, the tolerances for performing the comparisons (an absolute tolerance, a relative tolerance, a combination of both, and for arrays, whether the comparison is element-wise or with respect to a norm). In addition, the builder input specifies the frequency each particular probe is executed and debugging information printed (the compared data can be printed but is not recommended in general).

5.0 Future Developments and Plans

Finally, the following planned enhancements of the test harness are in design and partially implemented: 1) an additional tool to automate the insertion of the "include" lines into arbitrary Fortran code and generation of two builder input files; one that monitors no variables but provides builder input for all procedures in the application, and secondly one that monitors all input and output variables to all procedures of the application; 2) portable data formats for all checked data; 3) a C implementation of the template files, permitting C or mixed Fortran/C applications to be monitored, and 4) support for parallel SPMD MPI codes.

5.1 Portable Numeric Formats

There are currently two candidate libraries of procedures being investigated to support portable numeric formats; HDF5[6] and netCDF[7]. The plan is to select one of these and provide the libraries with the distribution to support one of these portable formats. With these formats, comparisons of results between different platforms will be facilitated; currently one has to use with formatted input/output which is not very satisfactory; unformatted input/output is processor-dependent and does not port in general between platforms.

5.2 Enhancements to the Support Tools

The very tedious and error prone aspect of installing the test harness in a large application is the insertion of the "include" lines and the preparation of the builder input files, specifying the variables is to be monitored, their properties (rank, shape), and the error tolerances. This represents a problem for two reasons; the first is it involves the preparation of many lines of files, and secondly, when they are wrong, the diagnostics by the builder tool, the compiler, or test harness itself are obscure and vague, because the errors can only be detected long after reading the input files where the error is present. To overcome this problem, a third tool has been designed and is partially implemented at the time of writing this paper. The tool will insert the correct "include" lines, modify the application source code to avoid the problems with labeled STOP, RETURN, and END statements, and IF statements. Secondly, it will create sample builder input files that can be readily modified by the user but will supply a list of all input and output variables for all procedures in the application. Thus, the new tool performs a sufficiently complete analysis of the application software to generate the need code and avoid the need to have redundant specifications from the user. Future versions of the tool will limit the building of these files to selected procedures specified by the user.

This tool will be written in portable Fortran 95 and available in all distributions. It essentially has to create a complete symbol table for the application, including the attributes of the variables needed to insert the test harness in the application code. However, this implementation is viewed as a reference version that will specify the functionality of the tool; efficient, compiler-specific implementations will likely follow that use the symbol tables generated by the compiler. The tool will be written using an API that will permit the development of symbol-table access procedures to any particular compiler, thereby taking advantage of the efficiency, robustness, and reliability of compiler-generated symbol tables that the reference version is unlikely to ever exhibit.

Once the design and reference implementation is complete, consideration of an implementation that uses a graphics-user interface and menus integrated into a Photran[6] environment will be considered. Such an implementation would then take advantage of the existing and upcoming tools supported by the Photran environment, permitting the development of an integrated maintenance environment for large application codes.

5.3 A C Version of the Test Harness

In its current form without the new tool described in 5.2, the dependence on Fortran is limited mainly to the template file used to create the application specific "include" files. The builder and includer tools restrict their knowledge of Fortran to its line continuation rules in the main. Thus, to create a version of the test harness for C and mixed Fortran/C codes, the template files need to be rewritten in C.

5.4 A SPMD Parallel Version of the Test Harness

The original idea for the test harness came from trying to debug a parallel SPMD code. The test harness's design was to support such code but has not been implemented on such code because the serial capabilities at the time were missing and were also needed for an SPMD implementation. Consequently, the plan is to revisit the development of code for such applications; the main issue is that each processor must create its own check data file and be able to read it in check mode. As with the serial version, the easiest and most frequently needed case is that each processor executes the code in the same execution sequence between the original and modified codes. Many codes behave this way and for these cases, a parallel version of the test harness is planned.

6.0 Implementations

Version 0.6 of the test harness has been installed and tested on the following platforms:

- Linux X86 using the NAG f95, GNU g95, and PGI pgf90 compilers
- Linux EM64T systems using the NAG f95 and GNU g95
- SUSE Linux and AIX using the IBM xlf95
- Cygwin using the NAG f95, GNU g95, CVF and Lahey Fortran compilers
- Windows XP using the CVF, Lahey, and Intel Fortran compilers

Version 0.6 of the test harness with documentation and installation tests is available on CD from the author.

7.0 Summary

A test harness for comparing versions of scientific computational software has been developed. Its main features including the comparison of floating point data by comparators that report only significant differences in the computed data. The criterion used to measure differences is based on relative and absolute tolerances specified by the user. The test harness is very effective at determining that modifications and enhancements to versions of application code maintain the same results as with previous test cases without performing tedious hand comparisons on pages of data.

References

[1] Brainerd, W. S., The F programming language, http://www.fortran.com/F, 2005

[2] Metcalf, M, Reid, J. K, and Cohen, M. Fortran 95/2003 Explained, Numerical Mathematics and Scientific Computation, 2004

[3] Smith, B. T., Creating a test data environment to detect errors in the code conversion process -- an overview, Version 0.6, Nov. 2005

[4] Smith, B. T., The test harness user's guide, Version 0.6, Nov. 2005

[5] PARAWISE, The Computer Aided Parallelization Toolkit, Tutorial Guide, Version 2.4, June 2004, http://www.parallelsp.com

[6] HDF5, http://www.hdfgroup.org/HDF5/, 2006

[7] NetCDF, http://www.unidata.ucar.edu/software/netcdf, 2006

[8] Photran, http://www.eclipse.org/photran, 2006