Final Report, Phase I

DOE SBIR Award No. DE-FG02-04ER84028

A Netaware Development, Support, and Maintenance Environment For DOE Numerical Libraries

Brian T. Smith
Principal Investigator

Numerica 21 Inc.

April 2006

Revised November, 2006
Revised December, 2006

Introduction

This is the final report for Phase I of the DOE SBIR DE-FG-04ER84028 awarded in July 2004. The report is organized under the topics of:

- The Problem
- Problem Analysis and Study
- The Approach For The Test Harness Project
- The Vision For The Test Harness Project
- Summary: A Critical Tool Prototyped
- Specifications of An IDE For The Test Harness
- Future Research Needed
- Appendices:
    A. An Abstract -- Test Harness Abstract
    B. An Overview -- Creating A Test Data Environment To Detect Errors In The Code Conversion Process
    C. The Test Harness User's Guide --

The Problem

A major software problem facing DOE and many other organizations is the reliable and efficient maintenance of large collections of numerical (or mathematical) and scientific application libraries. Associated with these collections are tools (e.g., version builds and regression testing tools) that support the collections, but typically there were no tools to support the porting of these collections to new computational environments, when the libraries were developed.

There is a need today to maintain both application and library software with their tools over a wide and evolving set of diverse computing environments. In the past the common view was that such software was discarded and written again for the new architectures since they were typically quite different from machine generation to machine generation. Today, however, it is recognized that the cost of this continually rewriting of software is generally prohibitive. Secondly, while machine architectures continue to change at the high performance end of the spectrum, they are relatively longer lived at the desktop end. At the desktop level, architectural changes are less frequent; the need for algorithmic drive updates of software. Thirdly, computational grids (basically, networked computing) are becoming more stable and commonly used so that the maintenance problem both is potentially facilitated by such grids and made more acute by the availability of increasingly diverse computing environments.

The focus of the effort reported here is numerical computation but the methodology envisioned and described below provides an approach for supporting the maintenance of both numerical and non-numerical software. In addition, the solution to the maintenance problem for scientific software described below is the development of tools and libraries to support the maintenance activity. Although some of these tools are numeric, the majority are not. The tools themselves must also be maintained and therefore the proposal is to develop a tool development environment for both numeric and non-numeric code. Furthermore, the development of tools results in the additional burden of maintenance of them, again over a large diverse collection of machines.

To address this issue, the approach proposed here assumes that the tools are also used to maintain themselves. This sounds complex, but in fact the development of the prototype tool described below is more robust because earlier versions of the tools themselves (initially the hand installed versions) were used to test and maintain later versions of the tools. This approach turned out to be invaluable in detecting early errors in the tool and permitting continuous rechecking of the behavior of the tools throughout the development of them.

A prototype tool, called the Test Harness, which is central to understanding and exploring the benefits and problems of this approach has been developed and demonstrates an effective approach that provides one critical tool that supports the maintenance of both numerical and non-numerical software.

In this study, another significant aspect became very apparent. The existing libraries and tools have been written in many different programming languages. New languages or enhancements to old languages to meet special needs in high performance computing, for example, the proposed new high performance parallel computing language Chapel, developed by the Cray Inc.(`http://chapel.cs.washington.edu`); the Unified Parallel C language, developed at UC Berkeley and LBNL (`http://upc.ubl.gov`); or co-array Fortran (`http://www.co-array.org`), are continually being developed. Thus, the tools developed must be language-flexible and language extensible. The problem is currently somewhat reduced by the move towards language standards that permit and support interoperability, both of the libraries and the tools. Our study recognized that it is essential that the future tools and libraries be designed with both interoperability of languages and the compatibility of implementations in mind. That is, the need for high performance may outweigh the desire for interoperability. Thus, the need for very high performance in some portions of a library or application may create incompatibilities and the tools developed must permit a balancing of the goals of high performance and interoperability.

Problem Analysis And Study

To begin the work, the PI visited Jack Dongarra at the Innovative Computing Laboratory at University of Tennessee Knoxville (`icl.cs.utk.edu/`) to obtain their understanding of the problem and the current tools and ideas of how to address the problem.

What became clear is that the emphasis of current efforts is on the development of new software for the new architectures and new algorithms that are appropriate to these architectures. Unfortunately, what we have seen over the past 20 years and more is that the old software, particularly application software, does not go away. There is a long term need to update and upgrade it to new platforms, although typically not the high performance platforms, but rather those platforms used for regular production, which are often desktops with commonly used operating systems and compilers. In a sense, the emphasis on the new machines (and new algorithms) makes the problem worse because every new machine or algorithm leads to even more software that must be maintained with no tools to do it. What is clear is that the much of the past-developed software does not disappear but has to be maintained for the applications that continue on new similar architectures.

What is missing in some sense is a debugger that is numerically literate and, as well, machine and language agnostic, concentrating on the scientific application and algorithms rather than the details of the system and language on which the software is implemented. For numerical and scientific applications, the computational values are the important results but in some sense are at the bottom of the ladder of information that is computed. What is at or near the top is the model itself that the software produces. What is desired is that the software, when ported to new machines, produces essentially the same model. Seldom do we compute a single number but rather we simulate something like a storm approaching the shoreline and our primary interest is in knowing whether the strength or direction of the storm changes when we make the code run faster or on a different machine rather than in determining that individual numerical results are different in the 6th significant digit, for example.

The current practice is to compare numbers, but such comparisons are too detailed as they are contaminated by rounding errors, and are different for each machine, compiler, and programming language. Comparing anything else is difficult and in many cases we do not have the software to make the comparison in this way. Consequently, the situation is that we have not developed the tools to do anything else at this point in time. At the far extreme is the symbolic debugger, which for scientific applications is not appropriate at all. A middle level is typically printed results, where comparisons of numbers affected by rounding errors are very difficult, but such comparisons are most often attempted. At the high level might be the comparison of results displayed by visualization tools, but there the imprecision of the common techniques for comparison such as “they look the same" is not very useful in locating errors and bugs in code and problems in porting the codes.

The Approach For The Test Harness Project

The approach taken was to create a tool that addressed the tedium of comparing individual quantities and at the same time, permitted the investigation and development of the high level comparisons of results, where useful and desirable. For the former role, the vision was to create a tool that, given a list of the variables that were to be examined, generated modest modifications of the application code and application-independent code templates, using a library of procedures that permitted the storing and restoration of data values, facilitated comparisons of past values with current values, and reported differences that were deemed "significant". Using the same tool for the second role, namely comparing models or high level results, some illustrative high level comparison procedures were developed and experimented with. The comparisons that were performed used norms of matrices and arbitrarily shaped arrays, which required access in some cases to non-local information (array extents, for example). Although not very algorithmically similar to what is needed for the comparison of models, it required the development of general techniques to access global data as would be needed for model comparisons but was implemented in a manual way with the initial experimental version of the tool. (For example, some techniques to perform high level comparisons for the solutions of differential equations appear in an article by W. Enright, JACM 195, pp.203-2006, 2006; a further discussion of problem appears in the upcoming proceedings of IFIP WG 2.5 Working Conference 9, Prescott, AZ, July 2006, published by Springer-Verlag.)

To demonstrate that a tool could be implemented and be useful, and to uncover where it was inadequate, a partially-automated prototype was designed and implemented in two stages. First, documentation describing the needed code insertions and their locations were prepared, with a brief analysis of what each insertion had to do. It was decided early to use the common language construct of internal procedures. This technique provided access to the application's data environment, limited by procedure boundaries, with special and controlled importation of the environment of other procedures via the particular language's external communication features. This is certainly possible in Fortran 77/90/95 with the features of Fortran 95 for external and module procedures. However, Fortran 90/95/2003 has a major restriction that an internal procedure cannot have internal procedures. To overcome this restriction, the monitoring of internal procedures has to be performed by code insertion which, in general, is problematic but techniques were developed to eliminate the problems. Despite such problems, it was decided to create documentation of what would have to be done to handle all potential language features used in any Fortran application.

Next, using this description, six different application codes were investigated. These included applications from a few tens of line to one of approximately eight thousand lines and one that involved parallel MPI. Five of these applications were then hand modified and tested to verify that the design was adequate and general.

The motivation to begin the second stage was that the hand changes were far too laborious, time-consuming, and extremely error prone. After spending many weeks hand-modifying one of the larger codes, it was much clearer how a tool could perform most of the application code modifications. The application code modifications were mainly code additions with some modifications to the added code. Consequently, the use of code templates was investigated and evaluated; the result of the investigation was the design and implementation of two prototype tools described below.

The two tools, called INCLUDER and BUILDER, perform the following tasks: from a set of general templates and input files specifying the particular procedures and variables to be monitored, a set of application specific templates are created. The application-specific templates represent synthesized code that monitors the particular variables (scalar or arrays) of intrinsic types at procedure entry points, procedure exits, and at any desired point in the code. The templates and the builder tool support all types of Fortran procedures (external, internal, module, main program, and subprograms with ENTRY statements, frequently used in application code written in Fortran). Once the builder creates the application-specific templates, they were included into the original application code using the tool INCLUDER.

The builder performed one other important task; it created both an active set of templates and an empty set of templates so that the INCLUDER tool could build a version of the application code with the test harness completely absent from the code.

These tools became invaluable. Indeed the test harness was installed in the builder tool, to continually check its results and performance with past results as it was developed. This demonstrated that the test harness could be used to maintain non-numeric as well as numeric code; thus, the test harness supports the maintenance of both the scientific applications and the tools to support these applications.

However, the two tools, used on the five test applications, pointed out further issues with the approach. Because the builder and includer tools are essentially text manipulators with only very limited knowledge of the application code language built into them, errors in the specification of what to monitor are very difficult to detect by the tools. Such detection and diagnosis are left to the compiler when compiling the application code with the test harness installed. Such detection is far away from the cause of the error and the diagnosis is with respect to generated code, not the application code, and is therefore mysterious to the user. This makes it difficult to debug the builder input.

The solution to this problem is believed to be the use of an integrated development environment (IDE), which uses information, such as the symbol table for the application code, created by a compiler. An example of such an IDE for Fortran is the Photran system from The University Of Illinois Urbana-Champaign, which uses the general IDE toolkit Eclipse (`www.eclipse.org/`). With the cooperation of specific vendor partners such as NAG (`www.nag.com/`) or open source partners using g95 (`g95.sourceforce.net/`), information from the symbol table for a particular application code would be used to generate the appropriate test harness code in the application-specific harness templates. In this way, the generated code would be correct by construction or the IDE that generates the symbol table diagnoses errors in the input source code. Because the IDE, such as that of Photran, would provide many other tools to support and maintain the scientific code and libraries, the Photran IDE or something like it provides a very attractive avenue for further development of the test harness.

The functionality of the proposed IDE is described below in more detail.

The Vision

As mentioned above, a prototype was developed, that immediately could and was used for code maintenance, showing the way forward to reach the goal where models are compared and not numbers. To address the needs now, the prototype tool compares numbers, but is implemented in a way that supports extensibility towards the dream. To see the vision, let us first review the current prototype tool, as it applies to the maintenance of scientific procedure libraries.

Suppose one has a library, typically large, that has been written for one or more machines. The maintainer is faced with the prospect of implementing the library for a new or enhanced compiler and is asked to create a new version for this library. Or, the maintainer is asked to port this library to a new machine/architecture/compiler. The past developers have documented (maybe) a set of tests that were performed on the library when it was first created but over time, new tests have been developed and accumulated to test the library.

The maintainer is faced with a daunting task. Assuming he/she can run the old tests, comparing the results (numbers, currently) with the new implementation is extremely time-consuming and error-prone (he/she misses differences that are significant because comparing hundreds of numbers by hand is boring).

Given the prototype developed to date, there is a partial solution. The maintainer inserts "include" lines into the original and modified source code of the library, typically four lines per procedure to be tested. These inserted lines can be left permanently in the code because the prototype tool, given the specification to do so, will synthesis a version of the code as if the inserted lines did not exist. Using the prototype tool to install the active version of the test harness, given a test data set, the original code is executed in a mode that dumps information about the values it produces from a given procedure. The modified or ported procedure compares its results with the past stored results and reports whether the results are "the same". If not the same, then the maintainer is then responsible to find out why. Additional probes can readily be added into the code to further investigate the source of the differences. Any existing probes can be readily disabled when the installed comparisons are no longer useful and then readily re-enabled at will.

The issue then is: What does the tool compare between the two runs? Certainly, it can and does compare individual values. Such comparisons, though, have to accommodate different rounding errors because either the code performs a different but "equivalent" computation or because the arithmetic properties of the machine are different. Thus, tolerances are required and supported by the current tool. But the current tool allows for another very important approach. Instead of comparing values, it can be user-extended to compare "models". Currently, for example, whole matrices can be compared where the maintainer sets criteria for measuring the difference (comparisons that measure the differences in individual elements, differences in regular sections and differences in norms are options in the test harness; more general comparisons must be specified by code supplied to the test harness).

At this point, maintainer-specified comparators are currently allowed so that if one knows how to compare the model represented by the numerical results, such models can be the basis of the comparison. For example, a comparator that determines how well a particular solution satisfies an equation (or whether the solution from the modified code is better than the solution from the original code) can be provided. It has been left for future development to create a library of comparator procedures to be used by the IDE alluded to above.

The vision then is to create a tool that allows the development of a set of comparators of results, ideally created by the author of the original code, and saved in a tool library for use and reuse by the maintainer, using the test harness. The model is that the scientific library is maintained with this monitoring specified within it. But, when the library is prepared for production use, the tool builds the application without the monitoring being present.

The tool would support multi-machine environments and thus can be used to maintain code across a grid of disparate machines. This would require considerable more development but in a sense the prototype is already grid-enabled; the tools have already been tested and installed on a wide variety of compilers and operating systems. A major enhancement that is needed is the use of a portable binary exchange format for the check data files, such as HDF5 (`hdf.ncsa.uiuc.edu/HDF5/`) or netCDF (`http://www.unidata.ucar.edu/software/netcdf/`).

Although the tool was designed and developed to debug a large parallel MPI application, the current tool has not been retested on such code. Further work is needed here. Currently, it is believed to be straightforward to use the test harness in the situation where the computation is statically located between the original code run and modified code run. Each process would create its own check data file to record its computation and then compare the data from the modified code with the corresponding data from the original code for each process. The more interesting and challenging application is to enhance the test harness so that it could accommodate changes in the number of processes, particularly the situation where one process (serial code) is compared with many processes. To do this, the test harness needs a way to index the data generated in the first run and have the second run read the previous data by reading the past data specified by an index. This would require the use of a shared data file and would probably only be feasible and effective for comparing results for major computational segments.

The tool is language agnostic, in the sense that it relies on a library of code templates. Although these templates are currently written for Fortran code, they can be rewritten for any language. The C version of the templates is currently being designed. Templates for other languages can also be written. For each new language supported, the major portion of the builder tool that must be updated is the part that constructs line continuation for the appropriate language. On the other hand, development of an IDE for the other languages could represent considerable effort, unless the tools developed by various efforts using the Eclipse environment, such as at the computer group at Los Alamos National Laboratory are used (C. Rasmussen, private communication).

A planned major enhancement to the test harness is a tool that uses compiler-generated information to select what is monitored and to insert the "include" lines into the code at the appropriate locations. Such an enhancement should be encompassed in an integrated development environment such as Eclipse. With such an integrated development environment, Eclipse's source version maintenance tools, compiler tools, and visualization tools could be completely integrated to provide a maintenance environment for scientific application codes and libraries that operate over a grid of machines, providing an integrated environment that smears over machine and architecture dependencies, providing reliable, portable, high-quality software.

Summary: A Critical Tool Prototyped

Although the code modifications can be performed by hand, the task is daunting for a large library. As stated above, two command-mode tools have been prototyped that make no use of compiler tools to generate the application-specific templates from the language specific templates. These templates represent code that calls procedures in the test harness module to perform booking keeping for the harness, dump data to the check data file, and read data from the check data file, performing the "identity" checking between the current results and past results. This tool is not intelligent in the sense that it takes its input, does little or no checking, and creates the needed templates. The result of running the tool on an application code is the generation of the include files needed to run the application code with the test harness installed in the application.

However, the design of a reference symbol table generator in Fortran to be used by the tool BUILDER has been created. This design accesses information needed by the tool BUILDER via an application-program interface (API) that will be documented. Preparing the API for symbol tables generated by vendor compilers will then provide access to stable robust systems and also to the many extensions implemented by the vendors. Also, this approach of using an API will facilitate the development of the test harness for C, and possibly other languages designed specifically for high performance computing, such as Chapel (DARPA supported HPCS project by Cray Inc.).

Specifications of An IDE For The Test Harness

An integrated development environment (IDE) is envisaged to support the installation of the test harness in an application code. It is envisioned that the IDE can read and digest (compile) the application code in its entirety and produce from this compilation a data base of information about each program unit, accessible in some form to the IDE, such as the API mentioned above. The information needed is essentially the traditional symbol table from the compilation process with variable-usage information such as that a particular variable is providing a value to the procedure, that a particular variable is defined with a value used elsewhere when the procedure is completed, or that a variable is used for both purposes.

Now let us assuming that this information is available by means of some callable tool, such as the API described above; the following list specifies the needed functionality to support the test harness.

1) The location of the following statements to permit the insertion of the specific include lines:
   a) The header statement (SUBROUTINE, FUNCTION, or PROGRAM statement) of a procedure.
   b) The first executable statement of a procedure.
   c) STOP and RETURN statements, including implicit STOP or RETURN statements appearing before END and CONTAINS statements.
   d) CONTAINS and END statements.
   e) READ, WRITE, and PRINT statements.
   f) at a user-specified place, representing a data probe

   The needed insertions are subprogram dependent, potentially different for main programs, external subprograms, internal subprograms, module subprograms, and

ENTRY procedure in the case of Fortran. For example, the main program needs to insert an include line that specifies the test harness is to be initialized and terminated gracefully, which is unique for the main program. Thus, at each of the above locations, the kind of procedure where each of these locations occurs is needed.

2) A list of variables in each class (variables used as references, variables defined, or both). For each variable, allow the selection of the following options:
   a) The place where the variable is to be monitored (procedure entry, exit, or at a probe point).
   b) If the item is an array, the array's bounds or the kind of array specifier (explicit, deferred, assumed-size, or assumed-shape specifier).
   c) If an assumed-size array, the expression that specifies the upper extent in the last dimension. (This is a requirement for both Fortran and C application code.) The expression must consist of accessible variables and constants; the IDE should verify this condition is satisfied.
   d) The tolerance to be used for comparison of floating-point entities (the absolute or relative tolerances, or both.
   e) When an array, the portion (regular section) of the array that is to be compared
   f) For all variables, the frequency of monitoring of the variable; for variables of discrete type, any differences are to be ignored.
   g) For character variables, the specification that letter case is to be ignored or that leading blanks in the comparison are to be ignored.
3) For derived type variables, specify the procedure to perform the comparison
4) For an entry, exit, or selected probe, specify a procedure to perform the comparison, using any accessible data to the procedure. This procedure would allow identity checks that involve collections of data for which there is no variable of derived type containing all the data. Such a procedure could also check that the data computed satisfies some condition, such as the computed data is the solution to a particular equation or satisfies some identity. It also would allow tests that determine whether the past solution and current solution are indistinguishable in a numerical sense in much more general ways that comparing individual data values. The procedure to do such checking would be provided by the maintainer of the application code or library but the procedure would be in a form that can be referenced and reused in other places. It would return a value that indicates whether the check was satisfied or not and thus be easily integrated into the test harness.
5) Provide attributes of the monitored variables to the builder, such as type, and names of procedures in which monitoring probes are to be inserted.
6) Check for name conflicts for names created by the builder for temporary variables needed to perform the "identity" checks. This is particularly important for monitoring internal procedures where code that shares the variable name space of the internal procedure is inserted into the internal procedure.

This IDE should use pull-down menus to specify the various options and request further information when needed for any particular option. The IDE should be capable of reading an existing application code with both builder "include" lines already inserted and a builder input file for the builder, and then allow the maintainer to modify what is monitored, checking for the correctness, where possible, of what is being specified.

The IDE should have several special options. One option is to specify without prompting that the test harness be installed in such a way that every input/output variable of every procedure is monitored. Alternatively, an option is needed to specify that every input/output variable in specified procedures is monitored, or every procedure is monitored but no variables application variables are monitored. This last capability allows the maintainer to verify the original and modified codes execute the same code (at the procedure level) in the same order and provides performance information for each monitored procedure.

A second capability should be a batch or command mode for the IDE; that is, it examines every procedure in, say, a library and places default monitoring into every procedure and then generates the specific "include" files needed for monitoring the entire library. Such a capability supports the use of the test harness in a make file to test code when it is installed in a new environment (a new compiler, operating system, or machine).

Future Research Needed

The vision above encompasses two major operations in the maintenance of scientific software. The first operation is essentially to verify that the same results, variable by variable, are obtained, comparing the code running in a correct or acceptable way with the result obtained from modified code. The modified code is typically different from the original code because it is:

1) being ported
2) being modified to enhance maintainability, readability, performance, or correctness, for example. It may be re-implemented on an updated or new compiler or with different options for the compiler, such as optimization flags.
3) being updated or translated to a new or different language.
4) being updated with new algorithms that replace existing ones or add new functionality. In the latter case, concerns about the correctness of previous functionality by the modified code can be checked.
5) being implemented in a parallel or threaded environment (where the original code is running in a serial environment).

As one progresses down the above list, the implementation of the comparisons between past and current results becomes more difficult and complicated. This leads to a second role for the test harness. This role is to perform the comparison of results at a much higher level. To support this role, the development of new comparison techniques is needed. For libraries, one might expect these to be generic or general techniques, but for arbitrary scientific codes, these comparison techniques are likely to be very application-specific and discipline oriented. What the test harness is really doing is comparing results and answering the following questions:

- Are the results the same or equivalent?
- Are the results solving the problem?

Currently, "equivalence" is checked by comparing individual computed quantities to known correct results. The test harness can be used to implement any comparison. For example, using a standard linear algebra example, does the computed solution to a linear system solve the linear system of equations or how well does it solve it, or do the current results represent a solution nearby the previous solution? In the linear equations problem, answering these questions involves much more than comparing computed results; it may require further computation to compute condition numbers or residuals to answer these questions.

For areas like linear algebra, comparison algorithms are well known. For other areas and applications, the techniques needed are both published in the technical literature (usually under the topic of sensitivity analysis) or the subject of current research (see W. H. Enright, Tools For The Verification of Approximate Solutions To Differential Equations, in Accuracy and Reliability in Scientific Computing, Edited by Bo Einarsson, SIAM(2005), pp. 109-121, Enright, W. H., JACM 195, pp.203-2006, 2006, N. Higham, Accuracy And Stability Of Numerical Algorithms, SIAM 2002, and B. T. Smith, Future Directions For Numerical Software Research, Proceedings of Working Conference 9, IFIP WG 2.5, July 2006, Prescott, AZ, Springer-Verlag, to appear 2007, also see `http://www.woco9.org`). But effort is needed to reformulate these results into algorithms that compare computed results to determine if such results represent the same solution. In particular, research is needed in the application areas to create these comparison algorithms and software. Given the test harness tool where comparison procedures can be provided in source text by the developer, the maintenance of numerical libraries and scientific application codes will be greatly facilitated by the requirement that such comparators be supplied with the library as a matter of course in the development of such libraries.

Appendix A – Abstract

A Test Harness TH For Evaluating Code Changes In Scientific Software
Version 2

Brian T. Smith
Numerica 21 Inc.
505-377-1455
University Of New Mexico
505-277-8338
Email: carbess@swcp.com

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 in the software. 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 a tool called the inserter for large application codes (typically four "include" lines are inserted per program unit to monitor variables at entry and an exit from a program unit). 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 future runs of the software are run to detect any significant changes in the results. A tool called the input_generator is available that creates an input file for the builder for one of two cases; one of the input file causes the test harness to monitor only procedure execution order and provide performance information of the application for the test case, and the second input file causes the test harness to monitor all "external" variables of the selected procedures during execution. For this latter input file, an "external" variable is a dummy argument, a variable in a common block, a variable accessed from a module, or a variable that is written to a file or read from a file.

The future runs of the software use modified versions of the application code with the test harness installed that represent enhancements to the code. 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 what data values are recorded and compared; writing and reading the past values and all comparisons are implemented by the test harness, with the coder specifying only the variables and the criteria diagnosing significant differences. The second file mentioned above generated by the tool input_generator is such a file; it can be readily modified to tailor what and how application code variables are modified.

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.

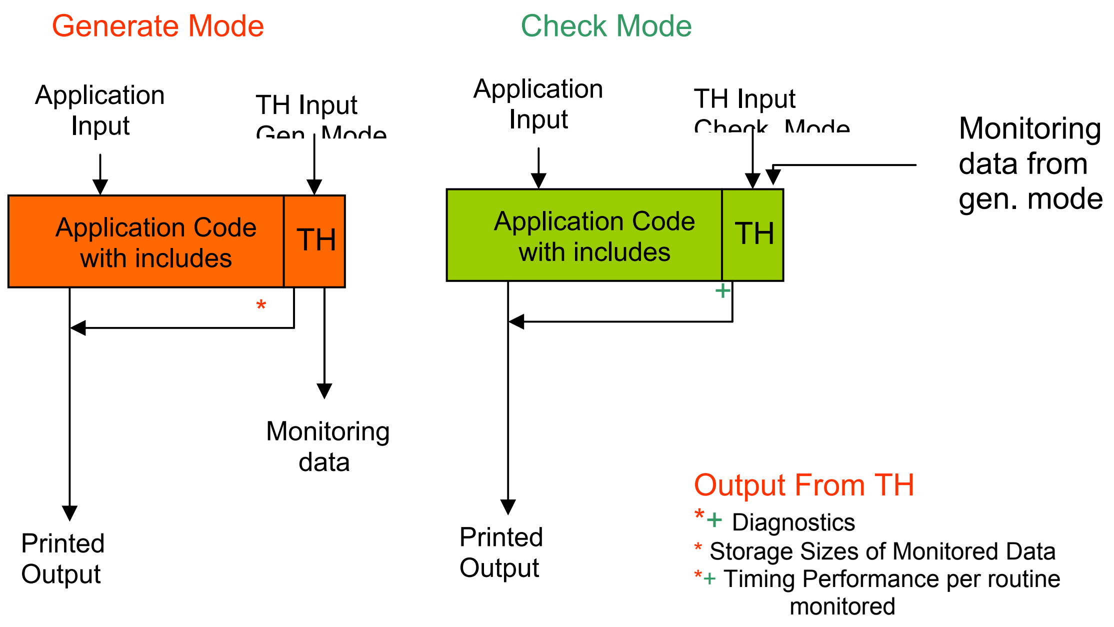
Control of the comparisons and what is compared is specified by input to the harness tool `builder`. The tool creates visible application-specific “include” files and a module with module procedures containing the comparator code that can be readily modified by the coder to handle special cases. In this way, unusual comparisons, say for combinations of specific elements of arrays or non-intrinsic derived types, can be coded into the specific include files as needed.

Figure 1 illustrates the modes of operation of the application code with the test harness installed. Figure 2 shows the kinds of “include” lines added to a main program which are required. Similar changes are necessary to any subprogram unit selected for monitoring. An “include” line is required for each additional probe point and each exit from a subprogram unit. 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.

Figure 3 illustrates the use of the test harness which has been tested on five or so application codes, including the builder tool. This figure illustrates the scenario of installing the test harness initially into an application. Enhancements such as, say, code changes to make the code Fortran 95 conformant, are made and each change is rechecked, comparing the results from the initial run. After some collection of changes, it is decided to monitor new sites. The test harness with the changes is reinstalled and the application code with the harness installed is rerun in generate mode. Again changes are monitored, running the code in check mode, as improvements to the code are made. At some point, a change is desired that changes the structure of the code or changes the results (a major bug fixed). Then, as before, a new comparison data set is required, created by rerunning the test harness code in generate mode without any rebuilding of the test harness, and the process continues until all enhancements are complete.

Once the test harness is installed in an application, it measures and displays the execution times and counts of the monitored subprograms and the check data file sizes. This information is useful in controlling the size of the check data files.

Figure 1: Operational Modes Of The Test Harness



Finally, planned enhancements are use of the HDF5 portable data formats for all checked data, a C implementation of the template files, permitting C or mixed Fortran/C applications to be monitored, and support for parallel SPMD MPI codes.

Figure 2: An Example – Instrumenting A Main Program

Original Application Code

```
Program main
   … Specifications

   … Executables

End program main
```

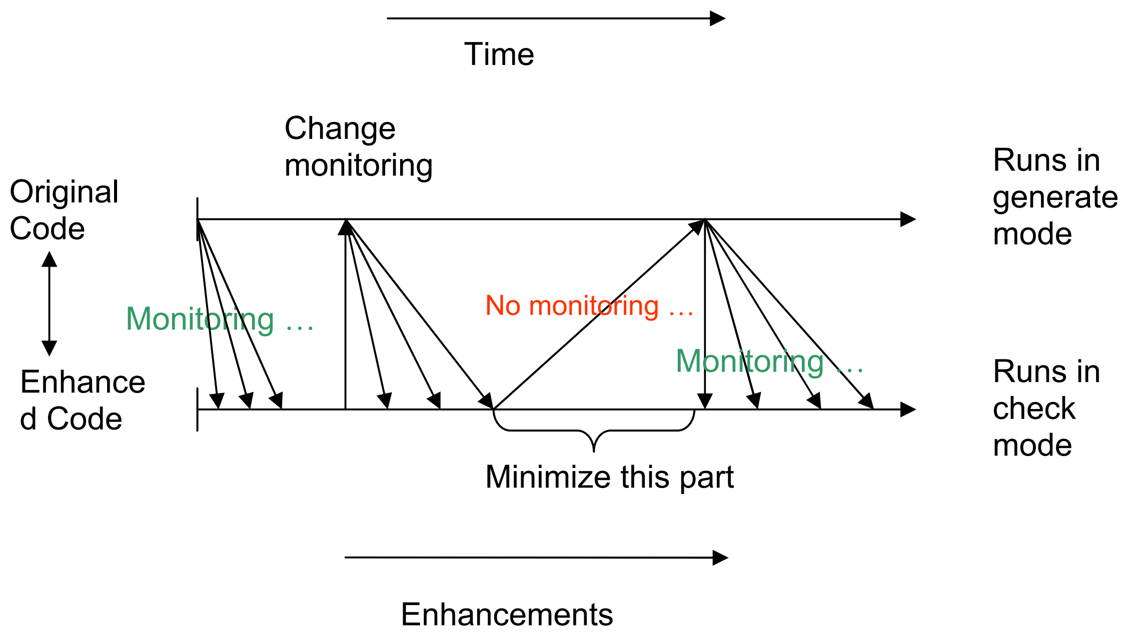
Application Code With TH Installed

```
Program mine
   INCLUDE "use_testing_harness_main"
   … Specifications
   INCLUDE "initialize_testing_harness"
   … Executables
   INCLUDE "write_output_and_finalize_all.mine"
End program mine
```

Plus similar INCLUDE lines for each:

- STOP statement
- Selected probe point

Figure 3: TH Usage Scenario



Appendix B – An Overview

Creating A Test Data Environment To Detect Errors In The Code Conversion Process

Version 0.6

Brian T. Smith

Numerica 21 Incorporated

12/14/2006 3:45 AM

Table Of Contents

1.0 Introduction
    1.1 Objectives
2.0 Structure Of The Software and Documentation
    2.1 Input File To The Test Harness
    2.2 Debugging Levels
    2.3 Frequency Of Recording Check Data
    2.4 Template “Include” Files
    2.5 The “include” Lines Inserted Into The Application Code
3.0 Testing The Harness
4.0 Uses Of The Test Harness
5.0 Restrictions
6.0 Planned Enhancements and Their Status

1.0 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 in the software. 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 (typically four “include” lines are inserted per program unit to monitor variables at entry and exit from a program unit). The instrumented application code, with an input file specifying the variables that are to be monitored, is analyzed by a software tool called the `builde`r. 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 future runs of the software are run to detect any significant changes in the results.

The future runs of the software use modified versions of the application code with the test harness installed that represent enhancements to the code. 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 what data values are recorded and compared;

writing and reading the past values and all comparisons are implemented by the test harness, with the coder specifying only the variables and the criteria diagnosing significant differences.

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.

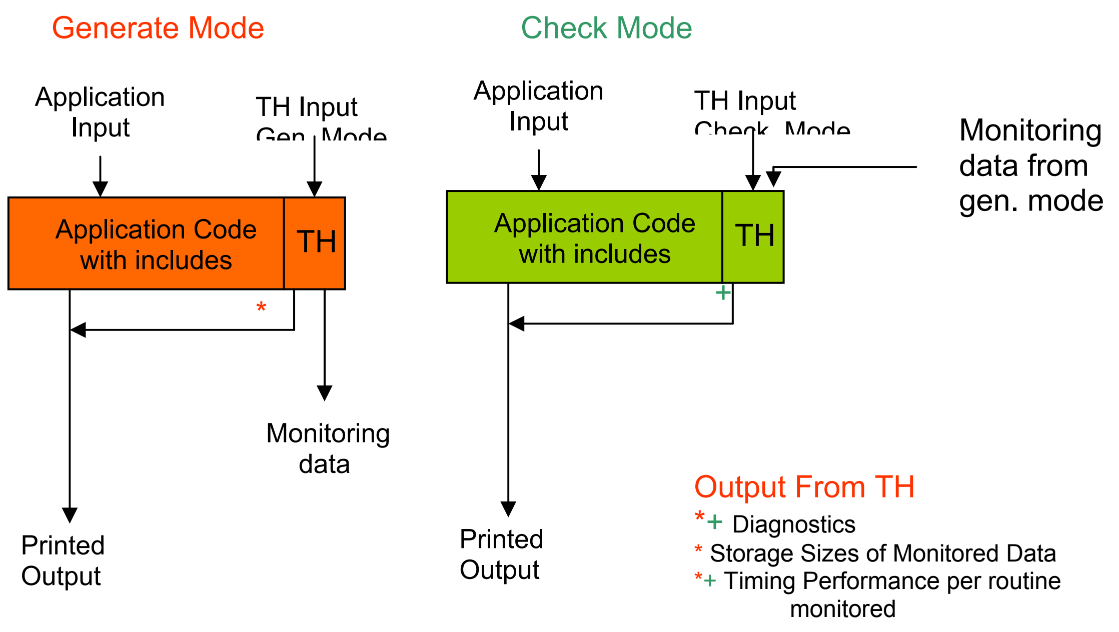
Control of the comparisons and what is compared is specified by input to the harness tool `builder`. The tool creates visible application-specific "include" files and a module with module procedures containing the comparator code that can be readily modified by the coder to handle special cases. In this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

Figure 1 illustrates the modes of operation of the application code with the test harness installed. Figure 2 shows the kinds of "include" lines added to a main program which are required. Similar changes are necessary to any subprogram unit selected for monitoring. An "include" line is required for each additional probe point and each exit from a subprogram unit. 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.

Figure 3 illustrates the use of the test harness which has been tested on five or so application codes, including the builder tool. This figure illustrates the scenario of installing the test harness initially into an application. Enhancements such as, say, code changes to make the code Fortran 95 conformant, are made and each change is rechecked, comparing the results from the initial run. After some collection of changes, it is decided to monitor new sites. The test harness with the changes is reinstalled and the application code with the harness installed is rerun in generate mode. Again changes are monitored, running the code in check mode, as improvements to the code are made. At some point, a change is desired that changes the structure of the code or changes the results (a major bug fixed). Then, as before, a new comparison data set is required, created by rerunning the test harness code in generate mode without any rebuilding of the test harness, and the process continues until all enhancements are complete.

Once the test harness is installed in an application, it measures and displays the execution times and counts of the monitored subprograms and the check data file sizes. This information is useful in controlling the size of the check data files.

Figure 1: Operation With The Test Harness



Finally, planned enhancements are use of the HDF5 portable data formats for all checked data, a C implementation of the template files, permitting C or mixed Fortran/C applications to be monitored, and support for parallel SPMD MPI codes.

Figure 2: An Example – Instrumenting A Main Program

Original Application Code

```
Program main
   … Specifications

   … Executables

End program main
```

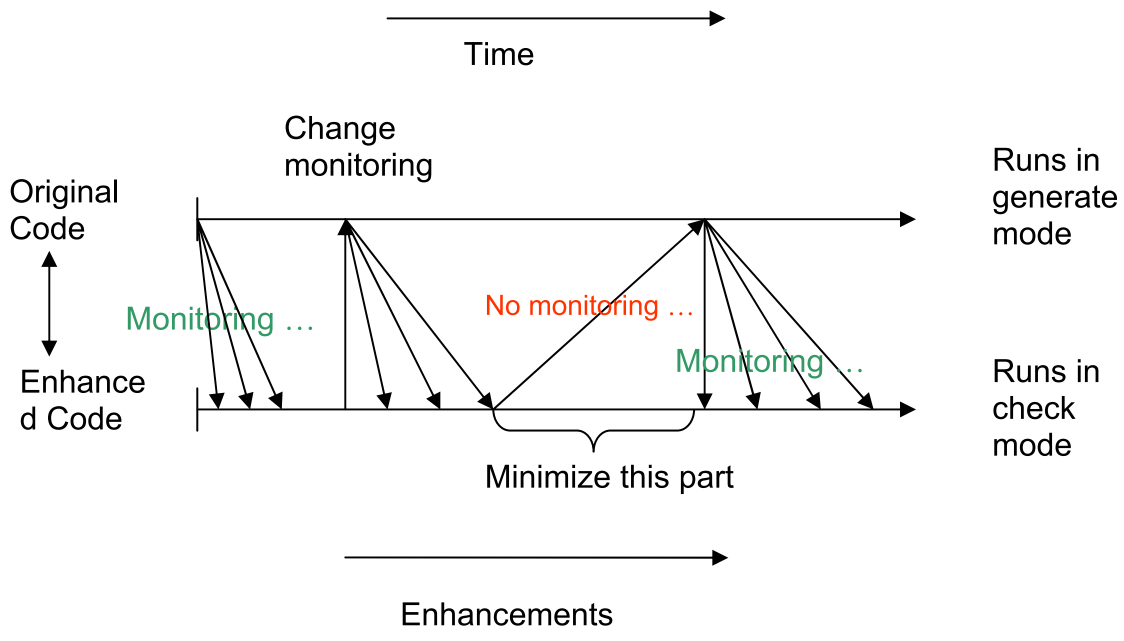
Application Code With TH Installed

```
Program mine
   INCLUDE "use_testing_harness_main"
   … Specifications
   INCLUDE "initialize_testing_harness"
   … Executables
   INCLUDE "write_output_and_finalize_all.mine"
End program mine
```

Plus similar INCLUDE lines for each:

- STOP statement
- Selected probe point

Figure 3: TH Usage Scenario



1.1 Objectives

The test harness is used to verify systematically that changes made to code consisting of a main program and many subprograms have preserved the computation. The current model for using the test harness is in a command-line mode; first, the application code is run to completion in a "data-generate" mode to generate data consisting of the values of selected variables at monitoring points; then, the application code, typically with a modified version of the source code, is rerun in a "data-check" mode where the current values of selected variables at the monitoring points are compared with the values read from the data generated by the previous run in "data-generate" mode. Implicitly, there is the assumption that the two runs are consistent in the sense that the generate mode and the check mode visit the monitoring points in the same order. An interactive version is being considered but at this time, only the design of the interactive test harness exists.

This test harness has been designed to have the following features and objectives:

1. The modification of application code is required only to the main program and those subprograms where data values are to be monitored. The modifications are essentially the insertion of "include" lines, except in cases of non-block IF statements with RETURN or STOP statements or of labeled RETURN, STOP, or END statements.

2. The additions or modifications are made to both the original code and modified code when testing the code. This is accomplished by:

    - Having the only additions to the original source code be the insertion of "include" lines.

    - Having the central part of the test harness be a module called `testing_harness`. This module has options, specified as character strings, that are read from standard input to specify the behavior of the test harness. These strings include:

        a) `generate` — run the test harness in generate data mode;

b) `check_and_continue` — run the test harness in check data mode, continuing after any discrepancy is found;
c) `check_and_terminate` — run the test harness in check data mode, terminating at the end of the probe where the first difference is detected;
d) `storage` — measure the sizes of check data file, generated by each probe, when in generate data mode; and
e) `performance` — measure the execution times of every program unit monitored in either generate or check data mode.

- Requiring changes to the test harness that are dependent on the application code to be made to the template "include" files and not the test harness code itself. These modifications can be made by hand but, because they are somewhat involved, time-consuming, and extremely tedious but regular, they are performed by a tool called `builder`. These changes are described briefly below.

3. The code can readily be run with the test harness activated or not activated. This is accomplished by having the tool `builder` create two sets of application-specific "include" files in separate directories; one set consists of Fortran statements that monitor application code data at specified probe points, and the other set consists of comment lines. When the application code with the test harness installed is compiled pointing to the active "include" directory, monitoring is performed. When the application code with the test harness installed is compiled pointing to the inactive or "empty include" directory, no monitoring is performed.

   When the "include" files contain active code from the test harness, the code should run in the original way, but executes the test harness in one of two modes, determined by reading data from standard input. The two modes are: generate mode, that is, generate and save test data in an unformatted sequential access file; or in check mode, that is, compare the current run's data values with those saved in an unformatted sequential file.

4. Use of the test harness requires a small number of modifications to the application code. The majority of the changes involve the insertion of "include" lines and are described in detail in 3.1 of the User's Guide. The decision to use "include" files restricts, however, the ways in which STOP, RETURN, and END statements are used in the application code. First, STOP and RETURN statements must NOT be the object statements in statement IF statements. If they appear as the object statements in statement IF statements such as:

```
label1 if( stop_condition ) STOP
```
   or
```
label2 if( return_condition ) RETURN
```

   these statements must be replaced by the block forms of these statements such as:

```
label1 if( stop_condition ) then
             STOP
       endif
```
   or
```
label2 if( return_condition ) then
             RETURN
       endif
```

   Secondly, if a STOP, RETURN, or END statement is labeled, the label must be removed and a CONTINUE statement, labeled with the removed label, must be inserted before such a statement. Unfortunately, there is an exception; if the END statement for a subprogram with internal procedures is labeled, the labeled CONTINUE statement must be inserted before the CONTAINS statement and the END statement label removed.

5. The “include” files are formatted in a source form that can be included in application codes written in either free or fixed source forms, used by standard-conforming Fortran 90, 95, and 2003 compilers. In addition, the synthesized code is in a form compliant for an F compiler and uses only Fortran statements supported by an F compiler.

6. The number of changes to the template “include” files to create the active “include” files is rather extensive. Instead of implementing these changes by hand, a tool called `builder` can and should be used to build the application-dependent active “include” files; although these active “include” files can readily be modified by hand to implement features not supported by the tool; such changes should be limited to the files `check_var_procedures.f90`, `timer_module.f90`, `th_parms_module.f90`, and `internals_*_frequency*`. The file `check_var_procedures.f90` can be modified to implement specialized checks for user-defined types or large data structures; the file `timer_module.f90` can be modified to implement a different timer for measuring execution times; the file `th_parms_module.f90` can be modified to specify different kinds for single and/or double precision, and to modify the default logical input/output unit DEFAULT_UNIT; and the logical expression in the IF statement in files with names of the form `internals_*_frequency*` can be modified to change the frequency of monitoring data. The last type of modification needs to be performed very carefully because changes in some of these file will effect monitoring all monitored variables, unless the modified “include” files are given different names and used selectively.

7. All application data are not required to be monitored; selected data checking is possible and encouraged. In fact, no variables of the application code need be monitored in any probe but probes can be inserted anywhere a complete statement can be inserted; in such cases, the test harness is verifying that the original and modified codes, run in generate mode and run in check mode, are executing the same subprograms in the same order, with the same frequency, up to the point where the code run in check mode terminates.

8. An extensible set of procedures is provided to check the identity or near identity of the monitored variables between current values and recorded values. Checking procedures for scalar and array values of all ranks and default kind intrinsic types, including double precision, are provided. For derived-type variables or other non-default kind intrinsic variables, the user needs create the checking procedures, insert them into the module `check_var_procedures_module` in the file `check_vars_procs.f90` in the directory `HARNESS_TEMPLATES`, and update the interface for generic procedure `check_var`. This is straightforward, following the approach taken for the intrinsic default types provided. The tool `builder` will create correct calls to these added `check_var` procedures without change to the test harness module itself.

9. The “include” line insertions and modifications to the application code are designed so that such insertions and modifications can be made by a compiler tool, given a list of procedures to be monitored. Currently, the insertions into the application code must be performed by hand but the creation of the application-specific “include” files with the information about what variables, their types, and their array properties is performed by the tool `builder`; such modifications to the application code and creation of the application-specific “include” file could be implemented more reliably and robustly by a compiler tool, knowledgeable regarding the syntax of Fortran.

10. The test harness is designed so that the template codes force runs in “generate-data” and “check-data” mode to execute the same code for each processor with the same MPI rank. Although the test harness has been installed in parallel code, further development and testing is needed before the code is ready for general use in parallel MPI codes.

11. The testing harness is designed so that variables can be readily monitored anywhere in the code execution sequence. Because of certain Fortran language constraints, application-code functions used in input/output lists or pure procedures can not be monitored with this test harness,

unfortunately, without changes in the application source code. See 5.0 for a further discussion of these restrictions.

2.0 Structure Of The Software and Documentation

The expected use of the test harness is first to install the test harness into a working application code, referred to as the original code in this document. Next, one makes changes to this code for various purposes, such as those outlined in Section 4.0 below, but with the goal of producing the same results on a test dataset as the original code. This second version is referred to as the modified version. The test harness then is used to verify the results are indeed the same for the given dataset(s). This process may be repeated, with different datasets testing different parts or features of the application code.

The test harness consists of a make file and a collection of template "include" files that are modified when the test harness is installed into an application code. The installation of the test harness into the original code (and the modified code if the test harness is not already installed) consists of the insertion of "include" lines into the main program and any program unit (maybe all of them) whose variables are to be monitored. The inserted "include" lines reference files which are the template files unchanged or the template files modified in various ways by the tool `builder`. Generally, the modified and unmodified "include" files are the same for both the original code and the modified code; this may not be true when the modified code has changed the control or data structures from the original code. For example, if the code is structured into different program units or data are moved from COMMON blocks to modules, these may require additional or different changes to the template files to make the monitored variables accessible to the test harness code. For example, the test codes in the directory `TEST_TEAM` are different in three major ways; the original code uses fixed source form, scalar operations, and `ENTRY` procedures. The modified code uses free source form, array syntax, and module procedures in place of `ENTRY` procedures. The use of module procedures to replace `ENTRY` procedures changes the organizational structure of the code but the execution sequence remains the same as far as the test harness is concerned (even though the "include" files are very different).

The documentation thus consists of this overview, a separate document which is a user's guide for the test harness, and a third document describing the operation of the tools `includer` and `builder` (to be completed). This overview document describes the input data files, debugging options, and operational flags for the test harness. It also briefly describes an installation test suite for the test harness that should be run when porting the test harness to a new compiler or operating system. This overview document concludes with the known restrictions on the use of the test harness and the status of a list of enhancements which have been either considered, planned, partially implemented, or completed at this time.

The separate User's Guide gives detailed instructions on how to install the test harness in an application code. The instructions include the building of a make file and the input for the two tools `builder` and `includer`. In application codes where the same program structure is used consistently, a rather small portion of these instructions is typically relevant to a given application.

However, with some compiler-knowledgeable tools (yet to be written), the installation of test harness in an application code could be made more reliable and robust. The changes to the application code, templates, and make files are extremely regular from a program structure point-of-view but very tedious and so error-prone. The data needed by the tool `builder` are a list of variables to be monitored, their attributes and location, and the criteria to be used to determine whether the results are identical or near identical. Typically, these variables are input or output variables at a particular point in the code, such as the entry to a procedure, the exit from a procedure, or at some point where there is a suspected problem. Once the user has indicated the locations of interest, an interactive tool with Fortran knowledge could generate the list of variables potentially referenced for their value and/or potentially changed from the previous probe point, requiring little or no further input from the user, other than how the comparisons for identity or near-identity are to be made.

2.1 Input File To The Test Harness

Standard input is used to establish values for the three test harness run-time modes, and two performance monitoring options. The three run-time modes are: generate mode, check mode, and continue mode. (Continue mode means that the test harness does not terminate on the discovery of any discrepancy between the generated data and the current data for the monitored variables.) The two performance monitoring options determine check data file storage sizes per application program unit and execution times per application program unit.

A single line is read from standard input on initialization of the test harness, which provides values for these test harness variables in order. For example, if the standard input unit reads the line:

```
generate
```

then the test harness is in generate mode, no file storage sizes are computed, and no performance information is accumulated. To run the harness in check mode with no storage or timing information created and termination after the first difference found, the standard input unit would read the line:

```
check_and_terminate
```

To run the test harness in the check and continue mode with performance information printed, the input line would be:

```
check_and_continue performance
```

Sample input files `harness_gen.input` and `harness_chk.input` are provided in the directory `HARNESS_TEMPLATES`.

If `storage` is specified and the code is in generate mode, the accumulated size of all data written from each application procedure to the check data file is printed at the end of the run. This includes the storage sizes for check data written by any probes within the application procedure. Because each ENTRY statement in a subprogram defines a procedure, the storage counts are for each entry procedure and not the entire subprogram in which the ENTRY statements appear. Also, the number of times each application subprogram is executed is also printed; this count is not affected by the recording frequency specified as input to the tool `builder`. If `performance` is specified, either in generate or check mode, the accumulated execution times and execution counts for each application subprogram unit are printed at the end of the run. Be aware that if the timer used has a long time interval between its ticks or the number of calls to the timer is large compared to the number of clock ticks measured, these run times may be inaccurate, contaminated by the timing errors. Also, because the insertion of "include" line is incorrect (needed ones are missing and it is rather difficult by program to determine reliably what is missing), the call counts for timing performance may be inconsistent. This erroneous behavior is likely because an input probe is specified but no output probe is specified or executed for a given subprogram or the subprogram terminates or returns to its caller without executing a test harness probe (that is, there is no "include" line for output monitoring before one of the RETURN or STOP statements in the program or subprogram unit). The tool `builder` checks certain sufficient conditions for valid performance times and prints the input and output (including stop probes) usage counts to help in assessing whether the performance measurements are likely to be valid.

2.2 Debugging Levels

There is a convenient mechanism to print, for a particular probe, the monitored values in ASCII of the test monitored variables on output unit DEBUG_UNIT by setting a flag specifying the debugging level. The levels are:

Debug flag with value	Action
0	No debugging output
1	Print all monitored variables from a specified input, output, or probe point

This flag can be specified independently for each input, output, or inserted probe (including the main program) for any procedure. The default in all cases is the debug flag set to 0. The input to the tool `builder` permits changes to these defaults. These flags should be used judiciously; they can produce a tremendous volume of output to the output unit `DEBUG_UNIT`.

2.3 Frequency Of Recording Check Data

There is an integer variable in each input, output, and probe procedure that specifies the frequency of recording of the values of the monitored variables. The default is 1, representing a recording frequency of 1, that is, the values of the monitored variables are recorded every time the particular input, output, or inserted probe is entered. A value of 100 represents a recording frequency of once in every one hundred times the procedure is entered; the larger the number, the less data is written to the check unit file, and the smaller the value, the larger the storage size of the data written to the check data unit. Other ways of limiting the size of the check unit storage sizes are possible; currently, other ways are left to hand changes to the code generated by the tool `builder` but may be implemented as enhancements in later versions of the test harness.

2.4 The Template "Include" Files

The directory `HARNESS_TEMPLATES` contains the "include" files referenced and inserted into application codes. To insert the test harness into the application code, these files are modified by the tool `builder` and inserted into a separate directory representing specific "include" files for the particular application code. A table in the document titled "Operation Of The Tools `builder` And `includer`" specifies the names and contents (or purposes) of the template "include" files inserted into an application code.

The source form used in these template files satisfies both the fixed and free source form rules of Fortran 90/95/2003. In addition, only syntax and constructs that are F compliant are used in these template files. The F compliance aspect represents a Fortran 90/95 code discipline that limits the source code to the simple and clean constructs of Fortran specified by the F language.

For purposes of documentation, the ideal names selected for the substitutable strings `#name#` and `#entry_name#` in the template file names are those of the actual names of the procedures in the application code. However, because of the way these names are used in the test harness, they are required to be unique among themselves and otherwise can be completely arbitrary. The capability to allow mismatches with the application procedure names is actually necessary, for example, when the names of two internal procedures in different scopes are the same name (permitted by the Fortran standard); however, the test harness requires such names to be unique in order to properly record execution times and storage sizes, identifying them with the appropriate unique name. See 3.0 of the User's Guide, which describes in more detail where these names are used.

2.5 The "include" Lines Inserted Into The Application Code

The "include" lines inserted into the application code use the keyword `binclude` by default; that is, the tools `builder` and `includer` interpret lines with this keyword by reading and manipulating the file named in the quoted string following this keyword. Both tools permit the specification of an alternative keyword to indicate an "include" line. This alternative keyword is case insensitive and may be `include`, to match what the Fortran language uses. By specifying this additional keyword as `include`, the application "include" files specified by Fortran INCLUDE lines will also be examined, manipulated by the tool `builder`, and included by the tool `includer`. If the additional keyword is not `include` or is not specified (the input line is blank), files specified by Fortran INCLUDE lines will not be processed by these tools.

3.0 Testing The Harness

Unzipping and untarring the file `th.tar.gz` creates a directory `DISTRIBUTION`. Within this directory are the following subdirectories and files:

`DOCS`	Documentation for the test harness in PDF format
`HARNESS_TEMPLATES`	Harness templates which represent the test harness, described in the documentation and are used by the tool `builder` to create the necessary "include" files for a particular application code.
`UTILITIES`	A directory containing various scripts and input files needed to run the test harness in an application code.
`TEST`	An installation test suite for the test harness.
`TEST_FAB`	A directory in which the test harness is used to access to evaluate the correctness of Fortran code modernization changes to a demonstration application code called `fab` (serial version) supplied by PSP. It solves 2D heat diffusion and conduction problems on a structured grid.
`TEST_JACOBI`	A directory in which the test harness is used to access to evaluate the correctness of Fortran code modernization changes to a demonstration application code called `jacobi` supplied by PSP. It is a simple 1D diffusion code using a point Jacobi solver.
`TEST_SIMPLE`	A directory in which the test harness is installed in one of the test suite program. This directory has been created by following the instructions of 2.0 of the User's Guide for the test program `fixed_ext.f90` in the directory `TEST`.
`TEST_TEAM`	A directory in which the test harness is used to access to evaluate the correctness of Fortran code modernization changes to a demonstration application code called `teamke1` (serial version) supplied by PSP. It is a 2D steady-state flow prediction code from the University of Manchester Institute of Science and Technology and employs a linear equation solver using the bi-directional LSOR method
`TEST_SHEAR10S`	A directory in which the test harness is to access to evaluate the correctness of changes to a parallel boundary element code `SHEAR10S` from the University of New Mexico. It was written by Ru Han, as part of her Ph. D. thesis, studying the flow of particles in slow-moving fluids.
`TH_LIB`	A directory containing the executable files `builder` and `includer`, and object and module files for the test harness module `testing_harness`. This directory has subdirectories for the various machine, operating systems, and compilers for which the test harness and its tools have been tested.

The following material describes the contents of the subdirectories `DOCS`, `UTILITIES`, `HARNESS_TEMPLATES`, and `TEST`. These four subdirectories provide documentation, tools, template files, and simple serial installation tests for the test harness, respectively. The other directories are applications (from simple to major test codes) in which the test harness has been installed and tested but are not described in detail in this document; they are documented in the "readme" files in each of the subdirectories. The directory TEST_SIMPLE is a directory like TEST but for one code. The directories TEST_JACOBI, TEST_FAB, and TEST_TEAM are like TEST but for one application code each, the simplest and easiest to follow is TEST_JACOBI, which is a main program of less than 50 lines.

The directory `TEST` contains test programs with names of the form *.f90 in subdirectories. The original test programs without the test harness installed are in the subdirectory `ORIG_TEST_CODE`; the modified versions of these codes in which the test harness has been installed with the required "include" lines inserted in the appropriate places are in the subdirectory `TH_TEST_CODE`. Also, in the directory

TEST, there are a collection of make files in the subdirectory MAKEFILES that compile and run these test programs for various compilers, operating systems, and machines. The test codes, the make files, and the modified templates in the subdirectories TEST_TH_INCLUDES (for an active test harness) and TEST_EM_INCLUDES (for an inactive test harness) of the directory TEST can be examined and compared with the original codes and templates to see what changes have been made. In addition, subdirectories with names of the form CP_TH_INCS_* and CP_EM_INCS_* where * is one of the suffices ext, int, mod, ext_int, mod_int, ext_int_noe, conflicts, mod_F, and int_F are, respectively, the active and inactive application-specific "include" files for the test programs with the corresponding suffices in their names. These application-specific "include" files have been built with the tool builder and are present to allow comparisons to check that a new installation of the test harness is working correctly.

These test programs check the installation of testing harness software, and include source files in free and fixed source forms, main programs with and without internal procedures, modules, module procedures, and procedures with ENTRY procedures. Also, included are programs to check F compliance where external procedures are not allowed

The subdirectory TEST_TH_INCLUDES of the directory TEST contains the "include" files created from the template "include" files in subdirectory HARNESS_TEMPLATES by the tool builder for a particular test. The subdirectory TEST_EM_INCLUDES in the directory TEST contains the empty "include" files, created by the tool builder, which when specified in the Makefile run the original codes with an inactive test harness. Also, in the directory TEST, are subdirectories OUTPUT_TH_INCLUDES and OUTPUT_EM_INCLUDES which contain the output results from the last time the tests were run.

To check that the test harness ports to your computer system, modify the make file or select one of the make files provided and type:

```
sh UTILITIES/run_all.sh ext
```

This will run one test, the test ext, and compare the results with previous results (actually it runs four codes, two codes are free source form and two are fixed source form). In particular, check the last line of the output files of the form *.out files, making sure this line indicates a successful test. The other tests can be run by changing ext to either int, mod, mod_int, ext_int, ext_int_noe, conflicts, int_F, or mod_F.

To run the tests in the other test directories, simply type make, after copying and modifying the appropriate Makefile from the directory MAKEFILES in a particular test directory.

4.0 Uses Of The Test Harness

The original design of the test harness was to support "modernization" code changes from dusty deck Fortran code to code that uses the safer and more general features in Fortran 90/95/2003, including changes that support code porting between precisions of floating-point arithmetic. This section describes this and other uses, and how to use the test harness to address these uses.

First consider the original purpose of the test harness. The process of code modernization in many cases requires no global restructuring, but introduces changes such as the use of modern control constructs (such as block DO constructs, CASE SELECT constructs, even IF constructs for some Fortran 66 codes), use of precision control features, use of generic intrinsic procedures rather than specific intrinsic procedures, use of array constructs rather than scalar ones, and use of allocated storage, data structures and derived types. For changes such as these, the following very effective process can and has been used on several large application codes:

- The test harness is installed in the original code (the code with the harness installed is fully compatible with fixed source code and compiles without diagnostics using a Fortran 90 compiler specifying fixed source form or free source form, depending on what the source form of the original code is). What is monitored is completely optional but experience with the test harness indicates that it is very useful to initially monitor the input/output variables of all or the major computational subprogram units. One keeps the original program with the harness installed so that what is monitored can be changed, depending on what problems are found. Because one needs to change occasionally what variables are monitored, the original program is rerun in generate mode to create different check data. A typical situation, particularly with large applications, is that the monitored check file becomes too large and must be made smaller by reducing the size and amount of monitored data. Ways to perform such reductions are summarized below at the end of this section and described in some detail in the User's guide. These ways include; the use of norms of arrays or particular elements rather than all elements of the array; use of the test harness variable `record_period` to limit the frequency of monitoring, for example, every 100th time (`record_period 100`). These latter two ways can be specified in the input file to the tool `builder`; another way is to modify by hand the test harness code generated by this tool.

- Run the resulting code in generate mode on representative data, saving the unformatted sequential file containing the monitoring data.

- Now, update and change the original code as desired, including changing its source form to free source form. The test harness remains in place and although the test harness may be changed and updated in the upgrading process (changing what variables or subprograms are monitored, for example), the structure of the test harness code will remain essentially the same.

- Now, run the code in check mode and make sure the results are the same. If not, investigate the causes for the differences; the differences may be caused by mistakes in the upgrading process, bugs in the original code, but also can be caused by computational instabilities in the code that will need to be addressed. If what is monitored is changed modestly, changes by hand can generally be made to the specific "include" files so that the original code can be rerun in generate mode to create a new or updated check data file and then the modified code is rerun in check mode to verify that no inadvertent change in results has occurred. It is highly recommended, though, to rerun the tool `builder` whenever any variables, their attributes, or comparison criteria are changed, rather than modifying the "include" files by hand; this tool builds the active "include" files very quickly, even for large applications.

In the code modernization process, the structure of the code may change in modest ways. For example, COMMON blocks may be replaced with modules, or ENTRY procedures replaced with internal procedures or module procedures, or external subprograms transformed into module procedures. Many of these structure changes do not effect the structure of the test harness code, as long as the same variables are accessible to a given subprogram unit before and after the modifications. Sometimes they do affect the accessibility of identifiers and therefore COMMON blocks or USE statements may have to be added to the harness's internal procedures; again, this typically does not change the structure of the test harness code. If it does, the "include" file names may have to change to point to the modified template files in the modified application codes; changes, such as conversion of COMMON blocks to modules and ENTRY procedures to module procedures, are very effectively performed for the test directory TEAM in the installation tar file, for example. (In the early development of the test harness, it had to be modified in rather simple ways to maintain accessibility to some of the needed variables; with the latest version of the tools and test harness, these changes may be unnecessary.)

For code modifications where the original and modified codes are structurally quite different, the test harness will have to be installed twice and great care will be required to ensure that the monitored data from the generate mode has the same structure as that read in the check mode. The recommendation for such cases is to perform the checks in stages so that at one key point the code structure changes but the comparisons are straightforward, then, reestablish a new test file with different monitored data and test harness procedures with the restructured code. Typically, when restructuring code, some variables and data remain stable; the test harness should focus on such variables when checking that the modifications made have not changed in the computation significantly.

Depending on the stage of debugging, the test harness can be effectively used to make sure known correct results are maintained. Again, like the major restructuring process above, careful selection of the monitored data can provide useful information in the code development and debugging process. Resetting the monitored data for new code cases or new test cases will likely be done frequently. For such uses, there is the need for IDE (integrated development environment) tools to quickly change the monitored variables and their locations. Such a tool is being designed and will be implemented as one of the proposed enhancements planned for the test harness.

With at most modest modifications, the test harness can be installed into parallel SPMD code using MPI. Under the model that each task records and checks its own data, using an unformatted sequential file per task, the test harness will work as is when there is exactly one task per processor in the parallel environment. provided the check data file is read and written from a disk, local to each processor. When there is more than one task per processor or the check data file is global to all processors, the OPEN statement for the harness test data file must use a file name which depends on the task id; such a change to the harness is trivial but needs to be tested in a parallel application code; the preparation of such tests is currently in progress and they will be placed in the subdirectory `SHEAR10S` in the directory `DISTRIBUTION`.

One other tool that performs a similar function to this test harness is a debugger. However, the test harness is designed for code that is less in the debugging stage and more in production. The test harness is no where near as general and flexible as a debugger, but has in fact been used effectively for debugging code where initially the location of a problem was unknown. It was used in a case where a code conversion project was converting hundreds of DO loops to array constructs. In the few cases where sum reduction loops were replaced by code using the Fortran 90 intrinsic SUM, the rounding errors were sufficiently different to cause the subsequent code to generate completely different results. Using the test harness, this was discovered quickly.

As a final point in the use of the test harness, one has to be careful not to monitor all elements of large arrays, particularly if such arrays are monitored frequently. If such is done, the size of the unformatted file will become enormous quickly. Selective judicial choice of the values to be monitored is recommended for production codes with large data structures that perform large amounts of computation. Another recommended approach is to modify the test harness's internal procedures so that the recording and checking of data is only performed at selected points in the computational process. To support this latter approach for simple cases, the value of the harness variable `record_period` can be increased from one to reduce the frequency of recording data to the check data file. Simple operations, such as reducing the frequency by resetting such variables, are performed by the tool `builder`. However, for more complicated ways of controlling when data values are monitored, because the user has access to the code of the active "include" files of the test harness, the code can readily be changed, for example, to record check data for the first 10 references, and after, say, the thousandth execution of the probe, whatever may be appropriate.

Thus, to address this last limitation in general production codes, it is critical that the user has access to the active "include" files of the test harness, even if these "include" files are generated with some automated assistance. Thus, it is recommended that the automated assistance always produce source code that needs to be compiled so that it can be modified as necessary by hand. The alternative would be to provide an automated mechanism to tailor the frequency of monitoring. Until such alternatives are implemented, this capability to modify the "include" files by hand is critical for the effective use of this test harness tool for large application codes performing a large amount of computation. The principle assumption, based on our experience, is that large codes require a very flexible tool, because they typically tax all resources to the extreme; general tools must be provide a mechanism to easily and readily adapt the monitoring to the needs of such large application codes.

5.0 Restrictions

Monitoring variables in a function, which is referenced in an input/output item list, is not possible using this approach without modifying the application source code. The function reference has to be removed from the input/output item list and replaced with a variable representing its value. Because the test harness performs input/output when a function is referenced, function references in input/output item lists create non-compliant Fortran code.

Monitoring pure functions or subroutines is not possible using the test harness unless the PURE attribute is removed. Consequently, this makes it impossible to monitor ELEMENTAL procedures, unless the ELEMENTAL attribute is removed and all references to them use scalar arguments.

The test harness creates variables and code for monitoring internal and F module procedures by inserting code segments into the scope of application subprograms. The created variables could possibly be the same as application variables and so create conflicts. The test harness tool `builder` permits the specification of a prefix for all harness variables and, separately, for all temporary variables, dependent on the variable names in the application code; the defaults for these prefixes are `th_` and `tp_`, respectively. However, input to the tool `builder` permits the specification of different prefixes, in case the application code has variables that begin with these default prefixes.

The test harness measures the computational time of each procedure it monitors. However, the measurement of the computational time is valid only when there is an input and output probe at each entry point and each return point of the procedures executed while the application is being monitored. That is, if one input or output probe is missing, all of the measured computational times are probably wrong. There is implicitly an input probe at the beginning of the main program, but all of the other input and output probes must be explicitly inserted into an application using “include” lines. The tool `builder` attempts to detect such invalid cases, but the detection is incomplete; complete detection requires more sophisticated tools that make sure every entry and exit point is monitored.

6.0 Planned Enhancements And Their Status

The following list is a record of the considered, planned, and implemented enhancements to the test harness at this version, currently 0.6. As the enhancements change their status for a given version, the status is recorded below. When a new version is created, the "Done" enhancements are deleted in the lists below.

- Create a version of the testing harness and the test code that will work with parallel MPI SPMD-style code. What about other parallel styles–eg: master/worker?

- Design and implement a specification for a GUI/Eclipse environment to:
  - Provide an automated environment for manipulating a large collection of subprograms.
  - Provide tools to determine candidate variables to monitor at any point in a program; for example, use program control and flow analysis to specify candidate variables that provide input to and output from a segment of code, probably limiting it to code segments with one entry and a modest number of exits (for example, 1).
  - Automate the changes required above to work with an application code.
  - Provide interfaces to run the test harness on an application code interactively.
  - Facilitate changes to the identity or near identity tests by suggesting criteria and making the required changes.
  - Facilitate disabling of the monitoring with the empty templates in automated way (specify the changes by routine and click a button to have the changes made).
  - Think of ways to specify an interactive mode for at least the checking mode. Allow interactive specification of the tolerances, for instance. Permit changes to the monitored data, which would recreate the check data file interactively.

- Instead of an unformatted check data file, generate a check data file in a portable exchange format – a procedure library for doing this is available as HDF5 from NCSA The advantage would be that porting an application to a new architecture from a working architecture could be checked without regenerating the check data file on the new architecture, thus adding an automated portability check.

- Consider writing a version of the test harness templates in C and versions of the tools `builder` and `includer` that generate C `#include` files and C probe procedures. Consider using the C interoperability features of Fortran 2003 to reduce the duplication of code for both a Fortran and C version of the test harness.

- Improve the NAG polish formatting options used on SHEAR10S, or try the Metcalf converter. Upgrade the code to use array operations instead of loops and make all arithmetic literal constants the same precision.

- Create and/or update the seven application directories to version 0.6 of the test harness. Details what is needed are provided below:
  - `TEST_APPLU` Create it from `WSB` work – work in progress
  - `TEST_FAB` Create it by copying from the `PARAWISE_TUTS` directory -- done
  - `TEST_FEMESH` Create it by copying from the `PARAWISE_TUTS` directory – work in progress
  - `TEST_JACOBI` Create it by copying from the `PARAWISE_TUTS` directory -- done
  - `TEST_TEAM` Create it by copying from the `PARAWISE_TUTS` directory -- done
  - `TEST_SHEAR10S` Create it by copying from the `RUHAN/SHEAR10S` directory – work in progress

Appendix C

The Test Harness User's Guide

Version 0.6

Brian T. Smith

Numerica 21 Incorporated

12/14/2006 3:45 AM

Table Of Contents

1.0 Introduction and Uses Of The Test Harness
- 1.1 Checking Data Values For Identity or Near Identity
- 1.2 Hints On How To Test For Near Identity
- 1.3 Extensibility Of The Checking Procedure
- 1.4 Installing And Verifying The Test Harness On Your Machine

2.0 Installing The Test Harness In An Application Code

3.0 Required Changes To Application Codes
- 3.1 Detailed Descriptions Of The Required Changes
  - 3.1.1 To The Application Main Program
    - 3.1.1.1 With NO Internal Procedures
    - 3.1.1.2 With Internal Procedures
  - 3.1.2 To Application External Subprogram Units Or To Module Subprograms
    - 3.1,2.1 With NO ENTRY Statements And With/Without Internal Subprograms
    - 3.1.2.2 With ENTRY Statements And With/Without Internal Subprograms
  - 3.1.3 To Internal Procedures
  - 3.1.4 To Module Program Units
  - 3.1.5 To Module Subprograms Of F Compliant Code
  - 3.1.6 Inserting Additional Probes
- 3.2 The Tool `builder`
  - 3.2.1 Description Of The Input To The Tool `builder`
  - 3.2.2 An Example Of A Standard Input File For The Tool `builder`
  - 3.2.3 Entries On The Input Line With The Keyword `#hvar#`
  - 3.2.4 Entries On The Input Line With The Keyword `#mvar#`
  - 3.2.5 Diagnostic And Other Information Printed By The Tool `builder`
- 3.3 The Tool `includer`
  - 3.3.1 Description Of The Input To The Tool `includer`
- 3.4 Creating a Make File That Uses The Test Harness In An Application Code
- 3.5 Reducing The Size Of The Check Data File
- 3.6 Obtaining Timing Performance Data

1.0 Introduction And Uses Of The Test Harness

This User's Guide lists uses of the test harness, describes how to install the test harness on your machine, and describes how to install the test harness into an application code.

In particular, this user's guide first describes the techniques used by the harness to check for consistent data values (either identical or near identical in deference to the real-life issue of non-portable rounding errors), and secondly, once you have decided what and how to check for identical values, to install the harness and permit the selected values to be checked at predetermined points, such as at the end of execution, but also at the entry points, exit points, and selected probe points of any selected program unit.

The test harness can be used for several purposes. These include:

- Checking that modifications and enhancements of a complex application code have not changed the behavior or results in unexpected ways. The purposes of such modifications might be:
    - code modernization, including code restructuring for better maintainability, readability, and portability
    - performance improvement and optimization, where the improved code is to give the same results as the original code, assuming one starts with robust solid code with a test data file that produces acceptable results.
    - code parallelization, comparing a serial version instrumented with the test harness with a parallel version instrumented with the harness. (This requires some enhancements to the test harness to run in parallel environments and to generate the data files in such a way that they can be used in a parallel environment).
- Code debugging in a special context of having a code nearly working and code modifications are being perform to improve the correctness, coverage, or stability of a code when compared with existing results of a known test case.

The accompanying document "Creating a Test Data Environment To Detect Errors In The Conversion Process – An Overview" gives the motivation and particular uses of the test harness software in more detail.

1.1 Checking Data Values For Identity or Near Identity

To verify that the data values of selected variables recorded in generate mode are the same as the values of the same variables in check mode, a generic procedure `check_var` is used to compare values for identity or near identity. For the discrete types such as integer, character, logical, or some derived types, we are most likely interested in identical results. But for floating point real and/or complex values, we are often interested in near identity because changes in the code or compilation options can change the order of the floating point operations and thus, because rounding errors may be different, change the values slightly. To handle this case, the generic procedure `check_var` has three optional arguments `abs_tol`, `ntype`, and `abs_tol` when its comparison variables are of type real or complex. This procedure actually has a fourth optional argument `lbnds`. This optional argument is not relevant to the comparison operation, but is used to specify the lower bounds of arrays of any type, when they have non-unit lower bounds. It is also possible to ignore differences in the values of variables of the discrete types; this is provided as a generalization for all types and is useful when differences of any monitored discrete-type variable are expected for some reason.

The optional argument `rel_tol` (usually a small integral value for scalar real or complex variables but may be a large value for a variable whose value is the result of a large number of floating-point operations) is used with epsilon for the floating-point kind to check for near identity with the following relative test:

```
abs(recorded_value - current_value) <= rel_tol*epsilon()*abs(recorded_value)
```

Thus, if `rel_tol` is, say, 2, and the recorded value is nonzero, the test requires the current value to be within 2 units in the last place of the recorded value. If `rel_tol` is zero, the comparison requires the `recorded_value` and the `current_value` to be identical to pass the check. The test has a flaw when the recorded value is zero, requiring an exact match – the `check_var` procedure should be called with the optional argument `abs_tol,` to specify an absolute error criterion to handle this case.

When the optional argument `rel_tol` is not present, or is zero and the optional argument `abs_tol` is present, the test for near-identity is:

```
        abs(recorded_value - current_value) <= abs_tol
```

which provides an absolute tolerance test. If both `rel_tol` and `abs_tol` are present, the test becomes a combination of a relative and absolute tolerance test, namely:

```
    abs(recorded_value - current_value) <= rel_tol*epsilon()*abs(recorded_value)+abs_tol
```

The second optional argument is `ntype`; it is used to specify whether the relative test for arrays uses an element or a global maximum for a comparative value; that is, if `nytpe` is absent or is a character value beginning with the letters "`ele`" such as "`element`", the comparison for near identity is:

```
        abs(recorded_array_value - current_array_value) <= rel_tol*epsilon()*
                                                     abs(recorded_array_value)
```

which compares each element relative to itself. Otherwise, the comparison is:

```
        abs(recorded_array_value - current_array_value) <=
                rel_tol*epsilon()*maxval(abs(recorded_array_value))
```

which compares elements relative to the maximum absolute element of the recorded array. That is, when `ntype` is absent or its first three characters are "`ele`", the comparison for each element is relative to the absolute value of the element, and otherwise the comparison is relative to the maximum absolute element value norm (`ntype` stands for norm type).

1.2 Hints On How To Test For Near Identity.

Which method of comparison is recommended? Initially, for arrays and scalars of all types, it is recommended you use the most conservative comparison, namely identity, that is, no relative tolerance or the `rel_tol` is set to zero. For most scientific applications, where an exact comparison is too sensitive, try next a small relative tolerance with `nytpe = "element"`, particularly for arrays. The messages diagnosing non-identity give the value the minimum relative tolerance that can be used to pass the near identity test for `nytpe = "element"`. They, as well, give the minimum relative tolerance for `nytpe = "global"`. If the message indicates a very large factor for `"element"` and a small relative tolerance for `"global"`, then a reasonable solution is to use `ntype = "global"`. If both the "`ele`" and "`global`" printed relative tolerances are very large, even huge numbers, the absolute test using the optional argument `abs_tol` may be appropriate, particularly if the norm is small. The diagnostic messages give a recommendation as to whether it appears to be appropriate to use the absolute test. If none of these alternatives is relevant or seems appropriate to the situation, the difference may be detecting an error in the application code.

An example of the diagnostic messages for differences in integer and real variables follows. The variables beginning with the prefix `tp_` are the values read from the check data file and the values of variables without this prefix are the values from the current run; note that the particular prefix used can be set in the tool `builder`'s input file but `tp_` is the default prefix.

```
        Message for integer data
```

```
 Variable test: variables named ivar and tp_ivar are unequal.
 The first difference occurs at element    34
 The values at this position are:           3           4

Message for real data

Variable test: variables named rvar and tp_rvar are unequal.
The first difference occurs at element    34
The values at this position are:  3.00000000E+00  3.00000095E+00
In order to be considered the same, rel_tol needs to be  2.66666675E+00
or larger using ntype:ele. Absolute tolerance is needed:  F
If ntype were global, the relative tolerance, for the variables to be
considered the same, should be:  2.66666675E+00
```

If a relative tolerance must be large to pass the near-identity check, the test harness has likely found a bug in the new implementation. Or, the original computation is unstable, indicating the original code is not robust and needs to be fixed. In either case, something is likely wrong and the code needs to be changed.

The recommendation to use the most conservative test first is for the case where the arrays are really collection of independent scalars, corresponding to element-wise computations. However, for many scientific applications using linear algebra computations, the global matrix element norm test is more consistent with the computation being performed.

1.3 Extensibility Of The Checking Procedure

The procedure `check_var` is a generic procedure overloaded for each of the five intrinsic Fortran types of default kind and double precision, and all ranks for arrays. For derived-type values or for non-default kinds, it is straightforward to extend the procedure `check_var` to such "new" types. The process is to write the comparison routine(s) in the style of those for the intrinsic types, possibly requiring extensions to the intrinsic function `pack` for scalar arguments, and inserting the extensions into the module `check_var_procedures` in the file `check_var_procedures.f90`.

For the new types or kinds where a new version of `check_var` must be implemented, it is recommended that a test be created in the subdirectory `TEST`. This can readily be done by enhancing one of the test program files in the directory `TEST/ORIG_TEST_CODE` and the corresponding files in the directory `TEST/TH_TEST_CODE` to include the new test, installing the new `check_var` procedure into the template file `testing_harness.f90` or `F_testing_harness.f90`, and then running the test programs of the installation test directory `TEST`, which test the new kinds or types.

1.4 Installing And Verifying The Test Harness On Your Machine

The installation package for the test harness is a tar file which, when untarred with the Unix command:

```
tar xvf th_distribution.tar
```

creates a directory `DISTRIBUTION` with the following files and subdirectories:

File or directory name	Description
`Readme`	A ReadMe file containing a description of the contents of the directory `DISTRIBUTION`.
`UTILITIES`	A directory containing the `builder` and `includer` tools, as well as other useful scripts and input files.
`DOCS`	A directory containing the documentation for the test harness which includes an overview and a User's guide (this document).
`HARNESS_TEMPLATES`	A directory of the templates which, when modified by hand or the `builder` tool, install the test harness in an application code.
`TEST`	A directory of simple test codes with the harness installed. These can be

	used as examples of how to install the test harness and are used as an installation test to establish local compiler options and system dependencies (there should be very few of these).
TEST_FAB	A directory where the test harness has been installed in two versions of the same fab code – a Fortran 77 scalar version and a modernized Fortran 95 code using arrays. It is a medium-to-large-sized code, solving the 2D heat diffusion and conduction problems on a structured grid
TEST_JACOBI	A directory where the test harness has been installed into two versions of simple 1D Jacobi code – a Fortran 77 scalar version and a modernized Fortran 95 code using arrays.
TEST_SIMPLE	A directory where the test harness has been installed into one of the test codes from the directory TEST
TEST_TEAM	A directory where the test harness has been installed into two versions of the same teamke1 code – a Fortran 77 scalar version and a modernized Fortran 95 code using arrays. This is a relatively large complex application, and is a 2D steady-state flow prediction code
TH_LIB	A directory with subdirectories containing executable files for the tools and object files for various operating systems and compilers.

The installation involves running as many of the test codes as desired, but in particular running the test in the directory `TEST_SIMPLE`, which represents a simple installation test for the test harness, following the instructions of 3.4 below. To run this test, copy (and possibly modify) the appropriate makefile from the directory `MAKEFILES` in directory `TEST_SIMPLE` into the directory `TEST_SIMPLE`, and then type:

```
make
```

correcting the system-dependencies until the test completes. The test runs one test program in generate and check modes, producing files with of the form `*.out` and `*.output` and specific "include" files in the subdirectories `SIMPLE_TH_INCLUDES` and `SIMPLE_EM_INCLUDES`. The tests are successful when an output line in the `*.out` file is either, for generate mode, the line:

```
Complete writing test output file.
```

or, for check mode, the line:

```
No differences found.
```

Previously obtained `*.out` and `*.output` files are provided in the directories `OUTPUT_TH_INCLUDES` and `BUILDER_OUTPUTS` respectively. The directory `CP_TH_INCS_simple` contains previously generated versions of the active "include" files and `CP_EM_INCS_simple` for the inactive (or empty) "include" files. The directory `UTILITIES` contains scripts that perform `diff` commands between the currently generated `*.out` files and those given in the directories `OUTPUT_TH_INCLUDES` and `OUTPUT_EM_INCLUDES`. The command `make` with no arguments runs these tests and actually compares the obtained results with past results stored in these directories.

The directory `TEST` contains 16 examples or tests in free and fixed source forms of the insertion of the test harness into simple programs, illustrating all the cases of main program, external functions and subroutines, module functions and subroutines, internal functions and subroutines, disciplined ENTRY statements where allowed, and probe points. For example, the test program `fixed_ext.f90` illustrates the insertion of the test harness into a Fortran 77 main program, external subroutine, and external function, with probe points inserted into each of these kinds of subprograms; the necessary changes to install the test harness into this test program can be seen by comparing the source test of the two files `TEST/TH_TEST_CODE/fixed_ext.f90` and `TEST/ORIG_TEST_CODES/fixed_ext.f90`. (This is the test code used in the directory `TEST_SIMPLE`.)

2.0 Installing The Test Harness In An Application Code

To use the test harness, the following step-by-step procedure is recommended (the subdirectories `TEST_*` are examples of following these instructions on different application codes):

- Create a directory, say `TEST_#PROG#`, where `#PROG#` is replaced by the name of the main program to be evaluated, in capital letters.

- In the directory `TEST_#PROG#`, create the following directories and subdirectories. Leave these directories empty; they are populated with files by the tool `builder` or by the instructions below:

  `BUILDER_INPUTS`, a directory
  `BUILDER_OUTPUTS`, a directory with subdirectories `DOS_FORMAT` and `UNIX_FORMAT`
  `CP_EM_INCS_#prog#`, a directory with subdirectories `DOS_FORMAT` and `UNIX_FORMAT`
  `CP_TH_INCS_#prog#`, a directory with subdirectories `DOS_FORMAT` and `UNIX_FORMAT`
  `OUTPUT_EM_INCLUDES`, a directory with subdirectories `DOS_FORMAT` and `UNIX_FORMAT`
  `OUTPUT_TH_INCLUDES`, a directory with subdirectories `DOS_FORMAT` and `UNIX_FORMAT`
  `#PROG#_EM_INCOUDES`, a directory
  `#PROG#_TH_INCOUDES`, a directory
  `TEST_OUTPUTS`, a directory
  `UTILITIES`, a directory

  where `#prog#` is the name `#PROG#` in lower case.

- Copy the original and modified application codes into the directory `TEST_#PROG#`. Use the file names `#prog#.f` and `#prog#_mod.f90` respectively for these programs.

- Make the "include" line insertions to the application codes as specified in 3.1; the make file assumes the application codes are called `#prog#.f` in fixed source form and `#prog#_mod.f90` in free source form. Select the modifications that are appropriate to the structure of the application code. The main program must be modified as specified in 3.1, but any or all of the subprograms may be modified as desired. The "include" line insertions in the two codes should be consistent (same program units and subprograms with "include" line insertions in comparable places).

- Create an input file for the tool `builder` as described in 3.2.1 and place in the directory `BUILDER_INPUTS`, naming the file `builder.input`. This input file must be consistent with the insertions of the "include" lines in the previous step; if there are inconsistencies, diagnostics from the tools `builder` or `includer` will likely appear on the error units. For the initial version of this file, it is recommended that as many as the defaults be used for the monitored variable attribute entries as possible; exceptions are:

  - For declared arrays or array items with non-unit lower bounds, specify the non-unit lower bounds with the 4th entry.
  - For arrays that are not completely defined, use the 9th entry to specify a section that is defined.

  - For floating-point variables (scalars or arrays), use the 5th and 7th entries to specify relative and absolute tolerances, respectively, for non-identity criteria when checking for consistency of the generate and check mode data values.

  Make sure the 4th and 5th lines of this file `builder.input` specify the "include" directories `#PROG#_TH_INCLUDES` and `#PROG#_EM_INCLUDES`.

- Create a make file `Makefile` for the application code with the harness installed by copying the make file from the subdirectory HARNESS_TEMPLATES and modifying it as specified in 3.4. The needed modifications will depend on the names of the files and subdirectories selected in the previous steps.

- After the modifications to the make file and application code are complete, execute:

  ```
  make #prog#_gen.out
  ```

  Fix all errors, detected by the tools `builder` and `includer`, and compilation problems found by the compiler until the code runs; often, the cause of the errors are mistakes in the "include" file names inserted into the application code or mistakes in the input to the `builder` tool, which unfortunately cannot be detected without an application code parser. When the make command runs without error, the application code with the test harness installed is run in generate mode, and thus generates a file named `$(MPN).test_output` in the main directory `TEST_#PROG#` where MPN is a macro in the make file; this file is an unformatted output file containing the values of all of the monitored variables. It may be large; check the output from the run in generate mode because it indicates the sizes of the data in storage units generated by each subprogram. See 3.5 for recommended ways to reduce the size of the check data file, if it is too large.

- Next, execute:

  ```
  make #prog#_chk.out
  ```

  which, depending on the make file `Makefile`, runs a modified version of the original application code, but this time in check mode; it compares the current values of the monitored variables in the modified application code with the values generated by original application code. The modified code will terminate whenever there is a significant difference in the monitored variables between the two versions. If the results are deemed "identical", the message indicating no differences found is written to the error unit `ERROR_UNIT` (the default is unit 6). Typically, though, corrections to the modified code or reducing the stringency of the near-identity checks in the test harness are required to reach this goal.

The main program must be modified as described below. No variables need be monitored from the main program but this would be unusual. The code included in the main program does monitor some variables local to the test harness though. These variables are used to check for consistency of the test harness environment. Unless the original and modified codes perform different computations or some input/output error occurs, non-identity checks for these variables should not occur. If no application variables are monitored and the two versions of the program complete normally, the test harness checks that the original code and the modified code are executing the inserted probes in the same order and frequency.

Typically, the monitoring of the main program or any other subprogram is performed at the beginning and termination of the main program and one or more subprograms. However, monitoring can be performed after any executable statement by inserting an appropriate "include" line into the application code as specified in 3.1.6.

The directory `#PROG#_EM_INCLUDES` has two roles. Taken as is and by setting of the make macro `INC_DIR` in the make file `Makefile` to this directory `#PROG#_EM_INCLUDES`, the modified program with the test harness installed can be run to verify that the program does not change its behavior when the testing harness is deactivated with the "empty" includes. This will verify that no inadvertent changes in the code have been made; this is the first role of this directory. The second role is to use the files from `#PROG#_EM_INCLUDES` selectively in place of files of the same name in `#PROG#_TH_INCLUDES` to deactivate selectively the monitoring of some of the application program's subprograms. This may be desirable either:

- Because no changes are made in these subprograms and so the variables need not be monitored;
- Because the sequential unformatted file may become too large (larger than the OS can handle) so that it may be necessary to monitor fewer variables in order to decrease the size of this data file without changing the template files drastically; or
- To perform a test to check if the compiler is generating different code, depending on whether or not the test harness included source text is present in the application code.

In the changes to the application code, it is assumed in the description below that no RETURN or STOP statement is the object statement of a logical IF statement, and no RETURN, STOP, or END statement is labeled. If either assumption is violated, then the application code needs to be changed in the manner specified in item 4 of 1.0 of the document titled “Creating A Test Data Environment To Detect Errors In The Code Conversion Process – An Overview”.

As noted above, the modifications to the application code, described below in 3.0, are only “include” line insertions into the application code which are described in 3.1.1 to 3.1.6. These insertions are different, depending on whether the program unit to be monitored is a main program unit, a subprogram unit, a module procedure, or an internal procedure, on whether the subprogram has ENTRY statements, or on whether the modifications are to be F compliant – select the modification instructions below appropriate to each case.

An alternative keyword that specifies the `builder` “include” lines may be specified to the input of the tools `builder` and `includer`. This capability allows the flexibility to either have these tools examine only the template “include” files needed by the test harness, or examine the “include” files of the application and the `builder` template files together. Note, although, that this keyword is case insensitive to both test harness tools, as is the “include” line keyword to the Fortran compiler. However, the built-in “include” keyword `binclude` used by the tools `builder` and `includer` is case sensitive; it must be spelled this way to be recognized by these tools and is used in all the examples.

For monitoring variables in function subprograms, it is assumed that the function is never referenced in an input/output item list. If it is and variables in such a function are to be monitored, the function reference must be removed in every input/output item list. In most cases, one can create equivalent code by the assigning the value of the function to a variable which is then used in place of the function reference. This or some other change must be made because, otherwise, the function with the test harness installed violates a Fortran constraint that an input/output item list must not cause input/output to occur (because the test harness creates input/output in the function).

For monitoring variables in pure subprograms (subprograms with the PURE attribute), this attribute must be removed. This is because, when installing the test harness into such programs, the installation results in code with input/output operations within the subprograms which causes the subprogram to be impure.

For monitoring variables in elemental subprograms (subprograms with the ELEMENTAL attribute), this attribute must again be removed, but this time other changes will likely be required. Where such a subprogram is referenced with non-scalar actual arguments, the reference will have to be replaced with code involving repeated calls to the subprogram with scalar arguments.

At first, there appear to be a lot of insertions into the application code. This is misleading in two senses: the description below is for arbitrarily structured code, and most application codes typically use one structure style throughout and thus only a few of the forms are selected below. Secondly, although creating lists of a large number of monitored variables in all subprograms is a daunting task, typical monitoring of the application code is restricted to a few subprograms and variables. The `builder` tool makes this part of installing the test harness less work because only in the input file to the `builder` tool are the subprogram names and the monitored variables specified.

3.0 Required Changes To The Application Codes

This section describes the required changes to the application codes. The changes are described in separate subsections, each subsection specific to whether the changes are to the main program or to subprograms. A summary of the required changes is given in document titled “The Operation Of the Tools `builder` and `includer` (to be completed)”. The purpose of the summary is to provide a broad outline of the actions that the tool `builder` performs.

3.1 Detailed Descriptions Of The Required Changes

Section 3.1.1 and its subsections describe changes to the main program of the application code. Sections 3.1.2–5 describe changes to the subprograms being monitored. Section 3.1.6 describes how to install probes into the application code at any desired point, provided the particular subprogram is being monitored already.

3.1.1 To The Application Main Program

There is a distinction between main programs with and without internal procedures. This distinction is necessary because the test harness uses internal procedures to read and write values of the accessible monitored variables. Inserting such procedures into main program units is different, depending on whether or not there are existing internals procedures in the main program unit. Internal procedures are used because they facilitate access to all identifiers in the monitored main program.

Also, there is a distinction between subprograms with and without internal procedures and with and without ENTRY statements. For subprograms with and without internal procedures, the issues are the same as for main programs. However, subprograms with ENTRY statements, regardless of any internal procedures they may have, essentially form multiple external procedures in the same scoping unit, thus requiring multiple input and output monitoring procedures that are also internal procedures, and therefore are quite different from subprograms without ENTRY statements.

Inserting the test harness into internal procedures adds further complications that must be considered., because internal procedures are not allowed in internal procedures in Standard Fortran. However, the approach used to handle these complications is the same as for F-compliant programs where internals procedures are not permitted at all

3.1.1.1 With NO Internal Procedures

- Insert immediately before the first specification statement after the program header, if present:

```
binclude "use_testing_harness_main"
```

- Insert immediately before the first executable statement:

```
binclude "initialize_testing_harness"
```

To monitor any variables at program initialization, insert after the above line:

```
binclude "write_or_compare_input_for.#name#"
```

where `#name#` is the name of the main program. This "include" line is optional and is only needed to monitor variables at program initialization.

- Insert immediately before the END statement:

  ```
  binclude "write_output_and_finalize_all.#name#"
  ```

  where `#name#` is the name of the main program. If there is a STOP statement immediately before the included line, replace the STOP statement with a CONTINUE statement. See item 4 of 1.0 of the Overview, if the STOP or END statement is labeled.

- Except for an explicit STOP statement immediately before the END statement, insert immediately before each STOP statement:

  ```
  binclude "stop_message_and_terminate"
  ```

  This "include" line writes a message and terminates the test harness gracefully; if application program variables are to be monitored before stopping, an "include" line, such as:

  ```
  binclude "write_or_compare_output"
  ```

  must be inserted before the stop message "include" line.

3.1.1.2 With Internal Procedures

- Insert immediately before the first specification statement after the program header, if present:

  ```
  binclude "use_testing_harness_main"
  ```

- Insert immediately before the first executable statement:

  ```
  binclude "initialize_testing_harness"
  ```

  To monitor any variables at program initialization, insert after the above line:

  ```
  binclude "write_or_compare_input_for.#name#"
  ```

  where `#name#` is the name of the main program.

- Insert immediately before the CONTAINS statement:

  ```
  binclude "write_output_finalize_no_contains"
  ```

  If there is a STOP statement immediately before the included line, replace the STOP statement with a CONTINUE statement. See item 4 of 1.0 of the Overview, if the STOP statement is labeled.

- Insert immediately before the END statement of the main program:

  ```
  binclude "finalize_main.#name#.internals"
  ```

where `#name#` is the name of the main program. If there is a STOP statement immediately before the included line, replace the STOP statement with a CONTINUE statement. See item 4 of 1.0 of the Overview, if the STOP or END statement is labeled.

- Except for a STOP statement immediately before a CONTAINS statement, insert immediately before each STOP statement:

```
binclude "stop_message_and_terminate"
```

This "include" line writes a message and terminates the test harness gracefully; if application program variables are to be monitored before stopping, an "include" line, such as:

```
binclude "write_or_compare_output"
```

must be inserted before the stop message "include" line.

3.1.2 To Application External Subprogram Units Or To Module Subprograms

There is a distinction between subprogram units with and without internal procedures, with and without ENTRY procedures, and subprogram units that are module procedures or internal procedures. These distinctions again are necessary because the test harness uses internals procedures to read and write values of the accessible monitored variables. Inserting such procedures into subprogram units is different, depending on whether or not there are existing internals procedures in the subprogram unit. As noted earlier, internal procedures are used because: 1) they facilitate access to all identifiers in the monitored subprogram; and 2) hide local variables of the harness procedures inserted into the application code.

3.1.2.1 With NO ENTRY Statements and With/Without Internal Subprograms

- Insert immediately before the first executable statement:

```
binclude "write_or_compare_input_for.#name#"
```

where `#name#` is the name of the subprogram. As well as providing the monitoring of the input variables to the subprogram, this "include" line establishes the name used by the tool `builder` to identity this subprogram.

- Except for a RETURN statement immediately before the END or CONTAINS statement, insert immediately before any RETURN statement:

```
binclude "write_or_compare_output"
```

See item 4 of 1.0 of the Overview, if the RETURN statement is labeled.

- Insert immediately before any STOP statement:

```
binclude "stop_message_and_terminate"
```

This "include" line writes a message and terminates the test harness gracefully; if application program variables are to be monitored before stopping, an "include" line, such as:

```
binclude "write_or_compare_output"
```

must be inserted before the stop message "include" line.

- With no internal procedures in the application external subprogram or module subprogram, insert immediately before the END statement:

  ```
  binclude "all_subprogram_pieces.#name#"
  ```

  where `#name#` is the name of the subprogram. If there is a RETURN statement before the included line, it must be replaced by a CONTINUE statement. See item 4 of 1.0 of the Overview, if the RETURN or END statement is labeled.

  With internal procedures in the application external program or module subprogram, insert immediately before the CONTAINS statement:

  ```
  binclude "write_or_compare_output"
  ```

  and insert immediately before the END statement:

  ```
  binclude "all_subprogram_pieces.#name#.no_contains"
  ```

  See item 4 of 1.0 of the Overview, if the END statement is labeled.

3.1.2.2 With ENTRY Statements And With/Without Internal Subprograms

The changes described below assume that the ENTRY statements are used in a rather disciplined way: namely, each ENTRY statement is either just after the specification part or is after a RETURN statement for the previous entry and there is no branching to code from one set of statements for an ENTRY point to another set. If such a discipline is not used, the complication is in selecting what variables to monitor at the RETURN, STOP, or ENTRY statements or at the probe points. Also, for a given `#entry_name#`, there may not be both input and output probes for all ENTRY procedures and the set of “include” lines to be inserted in the file `all_subprogram_entries.#name#` is difficult to specify and/or detect in general. Thus, such undisciplined code using ENTRY statements must be treated with some additional thought and care; the problem only is a difficulty in specifying in a completely general way the modifications required to correctly insert the test harness. Such modifications will have to be completely specified for the general tool, though; such a tool may abort in attempting to install the appropriate “include” lines in an application code where there is a completely undisciplined use of ENTRY procedures and require by-hand insertions of “include” lines into the application code.

- Insert immediately before the first executable statement (there must be one, if the assumed discipline is followed) the “include” line:

  ```
  binclude "write_or_compare_input.#entry_name#"
  ```

  where `#entry_name#` is the name of the subprogram (that is, the primary ENTRY procedure for the subprogram unit).

- Insert immediately before the RETURN statement before each ENTRY statement, the line:

  ```
  binclude "write_or_compare_output.#entry_name#"
  ```

  where `#entry_name#` is the name of the subprogram (that is, the primary ENTRY procedure for the subprogram unit). See item 4 of 1.0 of the Overview, if the RETURN or END statement is labeled.

- Insert immediately after each ENTRY statement:

```
        binclude "write_or_compare_input.#entry_name#"
```

where `#entry_name#` is the ENTRY procedure name (name on the ENTRY statement).

- Except for a RETURN statement immediately before an END or CONTAINS statement, insert immediately before any RETURN statement:

```
        binclude "write_or_compare_output.#entry_name#"
```

  where `#entry_name#` is the ENTRY procedure name appropriate for this RETURN statement. or the name of the subprogram, if the RETURN statement is before the first ENTRY statement. See item 4 of 1.0 of the Overview, if the RETURN or END statement is labeled.

- Insert immediately before any STOP statement:

```
        binclude "stop_message_and_terminate.#entry_name#"
```

  where `#entry_name#` is the ENTRY procedure name appropriate for this STOP statement. See item 4 of 1.0 of the Overview, if the STOP statement is labeled. This "include" line writes a message and terminates the test harness gracefully; if application program variables are to be monitored before stopping, an "include" line, such as:

```
        binclude "write_or_compare_output"
```

  must be inserted before the stop message "include" line.

- With no internal procedures in the application external subprogram or module subprogram, insert immediately before the END statement:

```
        binclude "write_or_compare_output.#entry_name#"
        binclude "all_subprogram_entries.#name#"
```

  where `#entry_name#` is the name of the previous ENTRY procedure and `#name#` is the name of the subprogram. See item 4 of 1.0 of the Overview, if the END statement is labeled.

  With internal procedures in the application external subprogram or module subprogram, insert immediately before the CONTAINS statement:

```
        binclude "write_or_compare_output.#entry_name#"
```

  where `#entry_name#` is the name of the previous ENTRY procedure and `#name#` is the name of the subprogram, and insert immediately before the END statement:

```
        binclude "all_subprogram_entries.#name#.no_contains"
```

  See item 4 of 1.0 of the Overview, if the END statement is labeled.

3.1.3 To Internal Procedures

- Insert immediately before the first specification statement (but after the internal procedure header) of the internal procedure:

```
        binclude "use_testing_harness_all"
```

- Insert immediately after the last specification statement of the internal procedure:

```
        binclude "internal_specifications.#name#"
        binclude "write_or_compare_input.#name#_code"
```

where #name# is the name of the subprogram.

- Insert immediately before the END statement of the internal procedure:

```
        binclude "write_or_compare_output.#name#.code"
```

  where #name# is the name of the subprogram. If there is RETURN or STOP statement immediately before the included line, replace the statement with a CONTINUE statement, and if a STOP statement, insert an unlabeled STOP statement after the included line. See item 4 of 1.0 of the Overview, if the RETURN, STOP, or END statement is labeled.

- Except for a RETURN statement before an END statement, insert immediately before any RETURN statement:

```
        binclude "write_or_compare_output_intmod"
```

  See item 4 of 1.0 of the Overview, if the RETURN statement is labeled.

- Insert before any STOP statement:

```
        binclude "stop_message_and_terminate_intmod"
```

  See item 4 of 1.0 of the Overview, if the STOP statement is labeled. This "include" line writes a message and terminates the test harness gracefully; if application program variables are to be monitored before stopping, an "include" line, such as:

```
        binclude "write_or_compare_output"
```

  must be inserted before the stop message "include" line.

3.1.4 To Module Program Units

- To a module program unit in non-F compliant code, no changes are required. The module subprograms in non-F compliant application codes are modified as described in 3.1.2

- To a module program unit in F compliant code, insert immediately before the first specification statement (but after the module header) of the module one of the following "include" lines, either:

```
        binclude "use_testing_harness"
```

  if the module has a PRIVATE statement without an item list, or:

```
        binclude "use_testing_harness_and_private_stmt"
```

  if the module does NOT have a PRIVATE statement without an item list. The module subprograms in F-compliant application codes are modified as described in 3.1.5.

3.1.5 To Module Subprograms Of F Compliant Code

F compliant codes cannot have RETURN, STOP, or END statements labeled; only CONTINUE statements can be labeled.

- Insert immediately after the last specification statement of the module subprogram:

```
binclude "internal_specifications.#name#"
```

  where `#name#` is the name of the module subprogram.

- Insert immediately before the first executable statement of the module subprogram:

```
binclude "write_or_compare_input.#name#.code"
```

  where `#name#` is the name of the module subprogram.

- Insert immediately before the END statement of the module subprogram:

```
binclude "write_or_compare_output.#name#.code"
```

  where `#name#` is the name of the module subprograms. If there is RETURN or STOP statement immediately before the included line, replace the statement with a CONTINUE statement.

- Insert immediately before any RETURN statement:

```
binclude "write_or_compare_output_intmod"
```

.

- Insert immediately before any STOP statement:

```
binclude "stop_message_and_terminate_intmod"
```

  This "include" line writes a message and terminates the test harness gracefully; if application program variables are to be monitored before stopping, an "include" line, such as:

```
binclude "write_or_compare_output"
```

  must be inserted before the stop message "include" line.

3.1.6 Inserting Additional Probe Points

To insert a probe at any point in the application code, there are two cases to consider. It is assumed in each case that the particular program unit has monitoring inserted into it already at either an entry or exit point. First, for a probe in a main program, external subprogram, or module procedure in a non-F compliant code, insert at the desired point an "include" line of the following form:

```
binclude ″write_or_compare_probe.#name#″
```

where `#name#` is a probe name, different from the name of any unit or other probe. Secondly, for internal subprogram or a module procedure in F compliant code, insert at the desired point an "include" line of the following form:

```
binclude ″write_or_compare_probe.#name#.code″
```

Also, in both cases, the input to the `builder` tool needs to be created to specify the probe and in particular, the variables to be monitored.

3.2 The Tool `builder`

The test harness tool `builder` reads the application code, annotated with the `builder` "include" lines described above in 3.1, and creates the specific "include" files from the harness template "include" files. The `builder` tool may be presented with the application code in one file or may be invoked many times on partial components of the application code. The tool `builder` creates and adds specific "include" files to the specified directory, containing the specific "include" files in `builder`'s input as the `builder` tool processes each file; if a specific "include" file by name already exists, it is reused; no specific "include" files, if they exist, are ever overwritten. This means the first version of a specific "include" file created is used in all cases; the later versions are expected to be the same but are never checked.

An standard input file to `builder` specifies the file name containing the application code, the name of the output file, the name of the directory containing the input harness template "include" files, the names of the two output directories to contain the active harness "include" files and the inactive or "empty" "include" files, and the "include" line keyword, used by the `builder` tool. Following these specifications are the values of the harness variables that specify such items as debugging levels and frequency of monitoring along with the names of the monitored variables for selected application procedures, their attributes, and the kinds of identity checks to be performed between the current and recorded values of these variables. Detailed descriptions of the possible parameters appear in 3.2.1 below.

The following section describes the `builder`'s options and the format of the `builder`'s input file.

3.2.1 Description Of The Input To The Tool `builder`

The first six lines of the standard input file specify the items in order as given in the following table. These lines as well as the remaining input lines are read with list-directed character input/output so that each entry is separated by blanks or commas. An entry that is to have one or more blanks or commas must be enclosed in single quotes or double quotes; if single or double quotes are used as delimiters and that

Line	Description	Example
1	The input file containing the application code with "include" lines	input.f90
2	The output file; the input file with any substitutions performed; there may be none.	output.f90
3	The input directory containing the harness template "include" files	../HARNESS_TEMPLATES
4	The output directory to contain the active harness "include" files	TEST_TH_INCLUDES
5	The output directory to contain the inactive "include" files	TEST_EM_INCLUDES
6	The `builder`'s "include" line keyword (case insensitive)	`include`

quote is required in the entry, it must be doubled. Also, blank lines are ignored and the first six lines are required; if one or more of these lines does not appear, the `builder` tool will fail with a diagnostic complaining about an unexpected hash-delimited entry.

The remaining lines of the standard input file essentially specify the values of the harness variables and the names and other properties of the monitored variables. The format of each line of the input is a keyword as the first entry (words surrounded by hash delimiters) followed by zero or more remaining entries, separated from each other by blanks or commas. Any entry that must contain a blank or comma must be enclosed by a pair of single or double quotes.

The entries are in nested blocks, starting with one of the keywords `#harness#`, `#main#`, `#entry#`, `#subprogram#`, or `#module#`. Following each of these keywords are variously the keywords `#internal#`, `#entry#`, or `#subprogram#` (modules have module procedures, designated with the keyword `#subprogram#`), and/or following the previous keywords are the keywords `#input#`,

#output#, #probe#. If a probe is present, it must be in the same block with the #input# and #output# specification and must be have the output specification for the block before it. Following each of these keywords is a name; the name is the name of the unit and must be unique amongst the set of all these names. The names correspond to the names used in the "include" lines, inserted into the source code. For the keyword #harness#, the name is always optional; for the keywords #input# and #output#, the names are optional, if the line specifies an input or output probe respectively in a program unit with no ENTRY statements; in all other cases, the name following the keyword is required.

Following these keywords are the keywords #hvar# (for harness variable) and #mvar# (for monitored variable). In all cases, the first two entries following these keywords are mandatory. The first entry is the name of the variable; the second entry after the keyword #hvar# is the value of the variable whose name is the first entry; and the second entry after the keyword #mvar# is the type declaration with attributes of the variable named in the first entry. There are no further entries for the keyword #hvar#, but there are six further optional entries for the keyword #mvar#. The details specifying the entries after the keywords #hvar# (for harness variable) and #mvar# are given in 3.2.3 and 3.2.4 respectively.

Indentation (blanks) before the keywords is allowed and encouraged to expose the block structure of the input, but are not required. But for one exception, the order of the blocks and lines within the blocks is immaterial; if some specification is repeated, the last occurrence is used. The exception is that any inserted probe specification must appear after the output probe specification for the same unit, and such an output probe specification must be present. Trailing comments beginning with an exclamation point are permitted at the end of any line and are ignored. Blank lines and comment lines (lines beginning with an exclamation point) anywhere in the input are ignored, except with the first 6 lines.

3.2.2 An Example Of A Standard Input File For The Tool `builder`

Before describing the detailed specifications and rules for the input, consider the following example:

```
input.f90                        ! Input source file
output.f90                       ! Output source file
../MODIFIED_TH_TEMPLATES     ! Dir. containing the TH templates
BUILDER_TH_INCLUDES      ! Dir. to contain the spec. include files
BUILDER_EM_INCLUDES     ! Dir. to contain the empty include files
include

#harness# testing_harness
   #hvar# debug_unit                     6     ! This is the default
   #hvar# error_unit                     6     ! This is the default
   #hvar# check_data_unit               21     ! This is the default
   #hvar# harness_input_unit            20     ! This is the default

#main# builder
   #output#
     #hvar# debug_level                   0   ! This is the default
     #mvar# input_file_name  "character(len=FILE_NAME_LENGTH) ::"

   #probe# after_input
     #hvar# RECORD_PERIOD                 1   ! This is the default
     #mvar# input_file_name  "character(len=FILE_NAME_LENGTH) ::"

   #probe# after_build_file
     #hvar# debug_level                   0   ! This is the default
     #hvar# RECORD_PERIOD                 1   ! This is the default
```

```
    #mvar# input_file_name  "character(len=FILE_NAME_LENGTH) ::"

  #internal# build_file
    #input#
      #hvar# debug_level                   0  ! This is the default
      #mvar# call_no           "integer ::"

    #output#
      #hvar# debug_level                   0   ! This is the default
      #hvar# RECORD_PERIOD                 1   ! This is the default

    #probe# output_line
      #hvar# debug_level                   0   ! This is the default
      #hvar# RECORD_PERIOD                 1   ! This is the default
      #mvar# loc_line              "character(len=LINE_LENGTH) ::"
```

This example is an application (actually the tool `builder`) which consists of a main program with some internal procedures, one of which is `build_file`, and a module with many module procedures. Despite the fact that this code has many other units, only the main program and one of its internal procedures is being monitored.

The first six lines specify the various file names, directory names, and “include” line keyword, used by the tool `builder`, as described earlier. The next line is blank, just to make a clear separation of the first six lines and the rest of the input, and is ignored. The rest of the input is in two major nested blocks; the first block, using the keyword `#harness#`, specifies several harness variables for the application-independent part of the test harness; and the second block, using the keyword `#main#`, specifies harness and monitored variables for the input, output, and probes of the main program and one of its internal procedures. Blank lines separate the blocks in the display to make the display more readable and are ignored by the tool.

The harness variables specified in the block `#harness#` specify Fortran logical units used by the test harness installed in the application code. It is strongly recommended that these unit numbers be disjoint from any Fortran logical units used by the application code. (The test harness also opens and closes logical unit 3 before the application code begins execution, and should not be in conflict with the application code, unless the application code assumes unit 3 is a pre-connected unit; this can be changed, if necessary, by specifying a different value than 3 for the named constant `DEFAULT_UNIT` in the file `th_parms_module.f90` in the directory `HARNESS_TEMPLATES`.)

Other blocks, such as ones using the keywords `#module#`, `#subprogram#` or `#entry#`, would specify information for application subprograms without entries and one with entries respectively; see the input files in the subdirectory `BUILDER_INPUTS` for other example `builder` input files.

Each of the blocks is similar. For example, most blocks specify a harness variable (keyword `#hvar#`) `debug_level`, giving it the value 0, which is the same as the default value, specifying no debugging output. Most blocks specify monitored variables as well. Note that the output probe for the internal subprogram `build_file` has no monitored variables and the harness variables use default value; this specification must be present, because there is a probe specified below it. Most unit blocks, except `main`, specify a block with keyword `#input#`; this block specifies the harness and monitored variables at the input point of the procedure in which the block appears; it need not have a name after it, if it is the only input probe (this is the only case for procedures with no ENTRY statements; for procedures with ENTRY statements, the `#input#` keyword must have a name after it to distinguish it from the other possible input probes). Similarly, for output, there is typically just one block. However, for probes, there may be many of them within a procedure and so all must be named. An example of a named probe is `after_build_file` in the internal procedure `build_file` of the main program.

Input and output probes may specify no harness or monitored variables. For the harness variables (#hvar# keyword), default variables and their values are provided by the tool builder. For the monitored variables (#mvar# keyword), input-specified variables, their declarations, and attributes are also printed by the tool. There are a collection of hidden monitored variables that generate or check an execution trace of the application code via the procedures specified in the input to the tool builder. If all procedures of the application code were monitored, a complete procedure trace of the application code would be generated and compared.

As a final item to observe, the use of the keyword #mvar# specifies a monitored variable; the entry after this keyword is its name (it must be the name of an accessible application program variable) and the entry after its name is its declaration. Both of these entries are required; there are six further optional arguments that specify how the current value and recorded value of the monitored variables are to be compared for being "the same". Because no other entries are supplied in all of the above cases, any non-identical current and recorded values cause warning messages to be printed by the test harness.

Details of the optional entries for the #hvar# and #mvar# keywords are provided below. Other examples of builder's input can be found in the directory BUILDER_INPUTS in the directory TEST for the 16 test codes. When a parser tool with a GUI interface is available, it will be unnecessary to give the declaration of the monitored variables because this information can be determined from the application code.

3.2.3 Entries On The Input Line With Keyword #hvar#

Lines with the keyword #hvar# in the #harness# block can specify the harness variables and values given in the table below. The default values for these variables are also given in this table.

Name	Description of Value	Default Value
check_data_unit	Fortran logical unit number connected to the check data file for writing in generate mode and reading in check mode.	21
debug_unit	Fortran logical unit connected for writing debugging information.	6
error_unit	Fortran logical unit connected for writing diagnostic message from the tool builder.	6
harness_input_unit	Fortran logical unit connected for reading input parameters to the test harness installed in the application code. This logical unit number must not exceed 29.	20

Lines with the keyword #hvar# in a #input#, #output#, or #probe# block can specify the harness variables and values given in the table below. The default values for these variables are also given in this table.

Name	Description of Value	Default Value
debug_level	The debugging level for the check variable monitoring procedure – level 0 is no debugging out, level 1 prints the current and check data values of all monitored variables for the particular probe, when the test harness is in check mode.	0
global_debug	The debugging level for the check variable monitoring for global variable checks – same specification as for debug_level	0
record_period	The frequency of monitoring for monitored variables – 1 means every time the probe is executed, 100 means every one hundredth time the probe is executed	1

3.2.4 Entries On The Input Line With Keyword #mvar#

A specification line with keyword `#mvar#` has nine entries, first three entries are mandatory (the first one being `#mvar#`) and the trailing six are optional. The following table describes these entries and gives their default values when the optional entries have the value `default` or are not present. (If one of the optional entries is to be specified, all earlier optional entries must be given; an entry consisting of the unquoted string `default` indicates the default value in the table below is used.)

Entry No.	**Description**	**Default value if `default` or absent**
1	The keyword `#mvar#`	No default
2	The name of the variable (no subscript, section, or component selector)	No default
3	The type and complete list of attributes as a complete attribute-oriented declaration, including the two colons (::), if a non-type attribute is used; if the variable is an array, the DIMENSION attribute with an explicit shape array declaration must be used.	No default
4	For arrays, the lower bounds for all subscripts of the array	(/1,1,…,1/)
5	For floating-point types, a relative tolerance for the comparison of the current and recorded values of floating-point variables	0.0 of the corresponding type
6	For arrays of floating-point types, the comparison for identity between the current value and the recorded value is element by element (`element`) or is a norm comparison (`global`). Only the first 3 characters in the entry are significant.	`element`
7	For integer and floating-point types, an absolute tolerance for the comparison of the current and recorded values of the variables; for the character type, it specifies whether the comparison ignores leading blanks (`ign_lb`), ignores any difference (`ignore`), or recognizes any difference (`recognize`) between the two strings; for logical type, ignore any difference (`ignore`) or recognize any difference (`recognize`).	For integer and floating-point types, 0 of the corresponding type; for character and logical type, `recognize`
8	The scope of the checking, whether `local` or `global`. When `global`, the first occurrence of the variable as a monitored variable in the `builder` input determines the parameters on how the comparison for the specified variable is performed, when the scope of any monitored variable of the same name is specified. `local` means the parameters for the comparison are determined, potentially differently, for each check of the monitored variable. `global` means the comparison parameters are to be the same throughout the application for all monitored variables of the particular name.	`local`
9	The variable that can appear in an input and output item list that either specifies a portion of an object (element, section, component) to be monitored, or specifies the extent of the final dimension in an assumed-size array dummy argument; eg. `A(:,:,1:N)` for a three dimensional assumed-size array `A`, where `N` is an accessible application-code variable.	The variable name

The optional entries must be specified in order, but the trailing ones may be omitted. An example is the line:

```
#mvar# array_a "real,dimension(2:10) ::" (/2/) 10.0 element 1.0e-9 local
```

This line specifies the variable `array_a`, which is a real 1-D array starting at subscript 2. The subscripts for the array have a lower bound of 2 in the first and only dimension. The comparison for "identity" involves a relative tolerance of 10.0, is performed element-wise, involves an absolute tolerance of 10^{-9}, and is to be local for this probe only, The final entry (no entry provided) defaults to the variable name `array_a`. See the test examples for other examples, particular for assumed-size arrays (see the file `BUILDER_INPUTS/conflicts.input` for the test named `free_conflicts.f90`).

3.2.5 Diagnostic and Other Information Printed By The `builder` Tool

The tool `builder` reads both the application code with "include" lines inserted and a standard input file specifying the values of harness and monitored variables. Because these two files are created separately, they may be inconsistent or incorrect. The tool `builder` attempts to diagnose anomalies where it can and continue execution; some errors cannot be recovered from and the tool terminates execution with a diagnostic message.

The nature of the errors detected by the tool `builder` can be categorized into several areas:

- Errors in the input specifications of the harness and monitored variables. Such problems typically cause the tool to terminate with a specific diagnostic message indicating the problem.

- Errors in processing the application-code after "include" lines have been inserted. At this stage, the tool has successfully read the harness and monitored variable specifications and has printed out two summary tables of the information in its data structures. The first summary table gives information about the application program structure as specified by its input about the harness and monitored variables. The data in this table should indicate a close match to actual application program structure; the format for this table is described below. The second table represents the values of the harness variables for each probe, the names and attributes of the monitored variables, and the parameters that determine how data values are compared; the format for this table is also described below. As a final diagnostic at this stage, the tool checks that for every input probe, there is at least one output probe, and for any output probe, there is an input probe. If this condition is not satisfied, the tool generates a message warning that the performance data will be erroneous, if generated. The check is a necessary condition for the consistency of the performance data but not sufficient; unfortunately, other errors and inconsistencies that can make the performance data erroneous and misleading cannot be diagnosed at this or any stage of the tool (such as a return from a procedure with no monitoring that is executed for some particular data values and not others).

- Errors in processing the application code. Most of these terminate the tool and provide as much information as possible as to what is wrong but cannot pinpoint the problem without completely parsing the application code (which is not done). Most often, the error is in using the incorrect "include" line, particularly inconsistencies between the "include" lines and the monitored variable input; hence, when something is diagnosed, check the two summary tables and "include" line insertions for inconsistent specifications. Just before the `builder` tool completes, it counts and prints the actual uses of the specified probes in the application code. As above, it checks another independent necessary condition for the valid generation of performance information, but it is not a sufficient check for this situation.

- Errors from the compiler diagnosing invalid code when compiling the application code with the test harness installed. This is likely caused by inconsistent or incompatible specifications of the names, types, and attributes of the monitored variables, or incorrect "include" lines inserted into the application.

An example of the form of the first table, the one giving the application program structure as determined by the harness and monitored variable input, is shown below. This table is the output generated by the tool `builder` with the test harness installed in the `builder` tool itself.

```
Builder's unit/procedure name list with attr. and ptrs:
 Legend: Position of Unit-Types and HI-Types

   Position     1      2      3        4        5       6
   Unit-Type  Main  Module Mod-Proc Int-Proc  NE-Sub  E-Sub
     HI-Type  Input  Output Probe

                  Name           Unit-Types  HI-Types  Units-Ptrs  HI-Ptrs
                        after_bf F F F F F F   F F T   N N N N N N   N N A
                     after_input F F F F F F   F F T   N N N N N N   N N A
                      build_file F F F T F F   T T F   N N N A N N   A A N
                         builder T F F F F F   F T F   A N N N N N   N A N
                     output_line F F F F F F   F F T   N N N N N N   N N A
```

This table uses a rather compact and cryptic format in order to relay the important information. The name field gives the name of the program unit/internal procedure whose status is being specified. The entries under `Unit-Types` and `HI-Types` are `T` (true) or `F` (false), indicating the type of unit. For example, in the first row of entries with the name `after_bf`, the `T` in the third column of `HI-types` (harness internal procedure type) entry indicates it is the name of a probe (legend notating the column position is given in lines 5 and 6 above). The name `builder` is a main program (`T` in column 1) and the name `builder_mod` is a module (`T` in column 2). The columns labeled `Units-Ptrs` and `HI-Ptrs` have entries N or A, indicating the pointer is non-associated(`N`) or associated(`A`) to a structure giving the harness and monitored variable information for the named unit. The second table below gives the relevant information that is associated with this named unit. The summary table given above is the one generated for the `builder` input given in 3.2.2.

An example of the form of the second summary table (giving the names and values of the harness variables and giving the names, types, attributes, and parameters for the identity checks on the monitored application variables) appears below. Again, this table is output from the tool `builder` with the test harness installed in the `builder` tool itself.

```
input name          : input.f90
output name         : output.f90
templates directory : ../HARNESS_TEMPLATES
specifics directory : BUILDER_TH_INCLUDES
empty directory     : BUILDER_EM_INCLUDES
INCLUDE line keyword: include
test harness prefix : th
temp variable prefix: tp


Harness/monitored variable names and attributes:
Legend:
    For the Nat (Nature) column
        main --> main program unit
        mod --> module subprogram unit
        mod_p --> module procedure
        int_p --> internal procedure
        sbwoe --> subprogram without ENTRY procedures
        sbwe --> subprogram with ENTRY procedures
        input --> input probe
        output --> output probe
        arbpt --> probe inserted at an arbitrary point
        ign_lb --> ignore leading blanks in character comparisons

Legend:
    default is different for each attribute:
        Default for: rel_tol: 0.0
        Default for: lbnds: (/1,1,...,1/)
        Default for: ntype: element
        Default for: abs_tol: 0.0 for numeric., leading blanks significant for nonnumeric
        Default for: scope: local


     Name      Nat  Kind Uses S_N hvars/mvars   name  rel_tol  lbnds   ntype  abs_tol  scope    var
value/decl
          th       harn  0    -1
                                hvars check_data_un
20
                                hvars    debug_unit
6
                                hvars    error_unit
6
                                hvars harness_input
21
    after_bf prob arbpt  0     1 hvars   debug_level
0
                                hvars  global_debug
0
                                hvars record_period
1
                                mvars     empty_dir default default default default default default
character(len=file_name_length) ::
                                mvars input_file_na default default default default default default
character(len=file_name_length) ::
                                mvars specifics_dir default default default default default default
character(len=file_name_length) ::
                                mvars templates_dir default default default default default default
character(len=file_name_length) ::
 after_input prob arbpt  0     1 hvars   debug_level
0
                                hvars  global_debug
0
                                hvars record_period
1
                                mvars     empty_dir default default default default default default
character(len=file_name_length) ::
                                mvars input_file_na default default default default default default
character(len=file_name_length) ::
                                mvars specifics_dir default default default default default default
character(len=file_name_length) ::
                                mvars templates_dir default default default default default default
character(len=file_name_length) ::
  build_file unit int_p
  build_file prob input  0     2 hvars   debug_level
0
                                hvars  global_debug
0
                                hvars record_period
1
                                mvars       call_no default default default default default default
integer ::
                                mvars input_file_na default default default default default default
character(len=file_name_length) ::
                                mvars loc_subs_entr default default default default default default
character(len=name_length) ::
                                mvars loc_subs_inpu default default default default default default
character(len=name_length) ::
                                mvars loc_subs_name default default default default default default
character(len=name_length) ::
                                mvars loc_subs_natu default default default default default default
character(len=name_length) ::
                                mvars loc_subs_outp default default default default default default
character(len=name_length) ::
                                mvars loc_subs_pres default default default default default default
logical ::
                                mvars loc_subs_prob default default default default default default
character(len=name_length) ::
                                mvars loc_subs_prob default default default default default default
character(len=name_length) ::
```

```
                                 mvars      new_unit default default default default default default
integer ::
     builder unit  main
     builder prob outpt  0     1 hvars   debug_level
0
                                 hvars  global_debug
0
                                 hvars record_period
1
                                 mvars     empty_dir default default default default default default
character(len=file_name_length) ::
                                 mvars input_file_na default default default default default default
character(len=file_name_length) ::
                                 mvars specifics_dir default default default default default default
character(len=file_name_length) ::
                                 mvars templates_dir default default default default default default
character(len=file_name_length) ::
 output_line prob arbpt  0     2 hvars   debug_level
0
                                 hvars  global_debug
0
                                 hvars record_period
1
                                 mvars      loc_line default default default default default default
character(len=line_length) ::
```

The table consists of 14 columns; the columns represent the unit name (`Name`), kind of probe (`Kind`), nature of the unit (`Nature`), usage count in the application seen in this run (`Uses`), subprogram number (`S_N`), whether a harness variable or monitored variable (`hvars/mvars`), the variable name (`name`), the relative tolerance (`rel_tol`), the lower bounds, if the name is an array (`lbnds`), the norm type used in comparison (`ntype`), the absolute tolerance (`abs_tol`), the scope of the checking specified in this procedure (`scope`), the variable used in an I/O item list, and the value if a harness variable and variable declaration, if a monitored variable (`value/decl`). For example, the last line of the table is specifying that in the probe called `output_line`, there are 0 uses (this table is printed before the application code is examined and so the count is necessarily 0), the probe is in unit numbered 2, the variable being monitored is `loc_line`, all of the parameters for comparison default, and its declaration is "`character(len=line_length) ::`".

The table format has rather long lines; the table would be better displayed in landscape format as the long lines are continued to the next line. This occurs in the above display for the column entry `value/decl` when the entry has a declaration with attributes appearing on the next line, such as "`integer ::`" or "`character(len=line_length) ::`".

The `Uses` column in the above table is always zero. The reason is that the table is printed before this item, which counts in the application code the number of "include" lines referencing this unit/probe, has been counted in the application code. However, the `builder` does print a table of uses just before it terminates as illustrated below for the test harness installed into the `builder` tool.

```
Final Usage Count Table:
    Unit Name   Input Count  Output Count
   build_file         1             1
      builder         1             1
```

For valid timing performance data accumulated by the test harness, all counts must be non-zero and the input counts should be 1.

3.3 The Tool `includer`

The tool `includer` is a tool to include the `builder` “include” files into a source file. After including the specified file, it modifies the “include” line into a comment line. The tool requires a standard input file that specifies, as well as other items, the input file, the output file with the “include” lines replaced by source text, the directory containing the “include” files, and the “include” line keyword used by the `builder` tool. The tool is used in the make file described in 3.5.

3.3.1 Description Of The Input To The Tool `includer`

The standard input consists of the following six lines:

Line	Description	Example
1	Starting logical unit number for reading the input and “include” files. This value MUST be larger than any of the units specified below.	`30`
2	Logical unit for error diagnostics – eg. “include” file missing	`6`
3	Unit for the output file with the “include” files included	`1`
4	The name of the input source file containing the “include” lines	`input.f90`
5	The name of the output source file with the “include” files included	`output.f90`
6	The directory containing the “include” files	`TEST_TH_INCLUDES`
7	The alternative `builder` “include” line keyword (case insensitive)	`include`

3.4 Creating a Make File That Uses The Test Harness In An Application Code

A template make file `Makefile` is provided in the directory `HARNESS_TEMPLATES`. This make file is designed for F90/F95/F03 compliant compilers – a second make file `Makefile_F` is designed to compile code with an F compiler. Create a working directory, say, `TEST_SIMPLE` in the directory `DISTRIBUTION`. Copy the template makefile `Makefile` from the directory `HARNESS_TEMPLATES`, and modify this template `Makefile` as follows to use it in a particular application:

- Set the names `original.f` and `modified.f90` to the names of the original and modified application programs. These names occur in several places in the make file.

- Set the make macros `APP_UC` and `APP_LC` to the upper and lower case names, representing the application. These are the names used for `#PROG#` and `#prog#` described in this document and are used several places in the make file as parts of file names, directory names, and other names.

- Set the make macros `MACHINE_OS` and `COMPILER` to the names of subdirectories in the directory `TH_LIB` used to designate your operating system and compiler.

- Modify the make macro S. specifying the directory separator used by the operating system. Slash is used for Unix, Linux, mingw, and cygwin under Windows XP, but for Windows XP, backslash must be used.

- Set the make macro A to TH or EM, indicating respectively that the test harness is active or inactive in the application code. EM is used only for testing purposes and to remove the test harness in compiled versions of the application code.

- Set the make macros F95_COMPILER, F95_LINKER, FIXED, and FREE to suit the compiler and operating system being used; FIXED, and FREE specify the compiler options to indicate the source code is in fixed or free source form, respectively.

- Set the make macros M, E, O, OBJ, and EXE to suffices and options used by the compiler to specify module auxiliary files, executable files, objects file, the option for object files and the option for executable files. The values set in the make file for these macros are appropriate for most compilers.

- Set the make macros FMT_TSTOUT, FMT_BDOUT, and FMT_INCS for the file formats generated for the application code standard output, builder tool standard output, and builder tool "include" file output. For Unix/Linux systems, the values should be UNIX and for some Windows systems, these values should be DOS. These options specify the formats of the output files, generated by the compiler, and are used to force the comparisons to be made with the same formatted files saved in various directories such as OUTPUT_TH_INCLUDES and CP_TH_INCS_#prog#.

Once created, the make command:

```
make #prog#_gen.out
```

creates the data file #name#.test_output where #name# is the name of the main program specified in the file BUILDER_INPUTS/builder.input. The make command:

```
make #prog#_chk.out
```

executes the modified source code, comparing the values of the monitored variables in the current run with those saved in the file #name#.test_output. Any values that are not "the same" (as specified by the builder input) are detected and diagnosed, and the program stops or continues until:

- An application program STOP statement is executed;
- If the string check_and_terminate is read from the harness input data file, the end of the subprogram or main program unit in which the non-identical values were detected is reached; or
- If the string check_and_continue is read from the harness input data file, the end of the program is reached or an application program STOP statement is executed.

The command

```
make
```

executes both of these commands in order, and compares the current results with the previously recorded results as described in 2.0.

3.5 Reducing The Size Of The Check Data File

The harness, once installed in an application code and the application code run in generate mode, computes the size of the data written to the check data unformatted file. These sizes are computed for each monitored application program unit and accumulate the data sizes for all probes (input, output, and explicit probes) used in the application program unit. When the application completes normally, these accumulated sizes are written out in a table at the end of the execution, giving the number of calls to program unit as well.

If the check data unit file is becoming excessively large, look at this storage size table for routines that are generating the most data. Consider reducing the frequency of the number of times the probes are called (see the entry `record_period` in the `builder` input file for the offending subprogram unit), number of probes or reduce the number of variables monitored. Also, an element or a section of an array may be specified, by using the ninth entry of the `mvar` specification, rather than monitoring a whole array. If these measures are not enough to reduce the dataset size to a satisfactory level, it may be necessary to modify the specific "include" file generated by the `builder` tool to replace the monitoring of complete arrays by monitoring representative values (such as subsets of individual array elements or norms of the arrays). Further enhancements to the `builder` tool are being considered and designed to assist this process. However, it is worth noting that on systems that will support it, check data files of the size of 10 gigabytes have been used with only a degradation of a few minutes in 30 minute runs.

An example of the output for one of the test files follows. The data sizes in this example are very small but the table below illustrates the format of the output.

```
Table of data sizes written to this point in
 storage units, typically 1 storage unit is 1 byte

Number Name                        Call count   Accumulated data size
   1   testing_the_harness             7                1058
   2   subr                            3                 410
   3   entry_ext_subr                  3                 410
   4   fcn                             3                 414
   5   entry_ext_fcn                   3                 414
   6   subr_fact                      29                2492
   7   fcn_fact                       29                2492
Total data size written to this point:         7690
```

The first number is the number of the routine; the names are those given in the input to the `builder` tool. The `Call count` column indicates the number of probes that wrote to the check data file from a particular program unit and `Accumulated data size` indicates the number of storage units written to the file from that program unit. A storage unit size is processor-dependent, but is typically a byte.

3.6 Obtaining Timing Performance Data

As a side benefit of installing the test harness in an application code, the test harness can generate, if selected, performance times for each monitored application procedure (see 2.1 of the Test Harness input Overview). Although these times may not be accurate because of interference from the recording or reading of data to/from the check data file, they are representative and useful in comparing relative performance with the test harness installed. Note that the execution times are measured for the application code and exclude the execution times of any monitored and called application procedure and of the test harness operations of reading, writing, and comparing. Also, the number of calls to the timers to accumulate a given entry in the table of execution times are provided, in order to indicate how the clock resolution error may be contaminating the listed times. Also, two versions of the harness are available; one uses a Fortran 90/95 standard timer that typically has a large clock resolution. A second version is provided that calls a C timer with a timer resolution of 1 micro second; however, it may not port to all systems.

An example of the output for one of the small test files follows. The times are in seconds; in this case, the clock resolution is so large and the execution time so fast relative to the clock resolution that times in this example are all zero but the table below illustrates the format of the output.

```
Table of execution times of application program units in seconds
Number Name                     Call Count No Accums  Accum. Time
   1   testing_the_harness          7         14    0.00000000E+00
   2   subr                         1          3    0.00000000E+00
   3   entry_ext_subr               1          3    0.00000000E+00
   4   fcn                     1          3    0.00000000E+00
   5   entry_ext_fcn           1          3    0.00000000E+00
   6   subr_fact              19         29    0.00000000E+00
   7   fcn_fact               19         29    0.00000000E+00
```

The first number is the number of the routine; the names are those given in the input to the `builder` tool. The "Call count" column indicates the number of calls to the particular program unit and "No Accum" indicates the number of calls to the timer to obtain the measured time. The column "Time" is the accumulated time in seconds for the particular program unit.