Automating the Debugging of Large Numerical Codes

Fredrik Manne
Svein Olav Andersen[1]

ABSTRACT The development of large numerical codes is usually carried out in an incremental fashion and over a long period of time. In this process, a segment of code might, for reasons of speed or accuracy, be replaced with a new code segment. If the resulting program contains a fault or generates a completely different result than the old one, it is crucial to find the place where the discrepancy first occurs. Finding this place might be a very tedious and time-consuming operation.
We present a framework called *comparative debugging* specifically designed to aid the programmer in debugging such situations. In this setting, the programmer can run both the old and the new program simultaneously, while comparing the execution flow and the values of certain variables in the two codes. If any discrepancy is discovered between the two programs, this is automatically reported back to the user who can then take appropriate action. The presented ideas have been implemented in a debugging tool.
We believe this to be a simple yet powerful way of automating the debugging process that can easily be incorporated into existing debugging programs.

1 Introduction

It has been estimated that 25-50 % of the total cost and time in system development may be spent on software testing and debugging [Boe81, Zel78], and as much as 95 % of the time spent on debugging may again be spent on fault-location [Mye79]. Therefore it is important to automate and speed up the process of locating where an error occurs in the code.

For this purpose programmers apply debugging tools such as the generic Unix debugger *dbx*, the GNU debugger *gdb* [SP94], and other vendor-specific debuggers. These tools let the programmer keep track of the state of the program and the values of certain variables while stepping through the execution. Even with these tools the debugging process can be time consuming since the programmer must inspect values manually to determine

[1]Dep. of Informatics, University of Bergen, N-5020 Bergen, Norway, {fredrikm,sveinoa}@ii.uib.no

where an error occurs.

In order to automate the process most debuggers can set conditional breakpoints (watch-points). In this way, the debugger stops only when some condition is evaluated to be true. For example, one might know the expected range of a variable. However, if the computation is "almost" correct such information will be of little help in determining where the error occurs.

Program slicing [Agr91, Pan93, Wei82] is a way of determining which lines of code affect a certain variable. Thus if one knows that a certain variable is incorrect this method gives an overview of the other variables that affect it.

One must, however, know in *which* variables the error occurs and also *when* it first occurs. An incorrect end result might be the result of a long chain of events in the program. For this reason execution backtracking [Agr91] has been suggested. This method lets the user backtrack through the execution of the program recreating the different states of the execution. For memory intensive programs with long execution time this is not a practical procedure as it would require storing vast amounts of data. A limited time window, in which one can backtrack will save memory but might not be sufficient to backtrack to where the error originates. Moreover, this process still requires the user to inspect the program manually.

The motivation for the current work comes from debugging large time-stepped simulation codes. Examples of such codes are fluid dynamics computations, oil reservoir modeling, weather forecasting, etc. These codes have in common that they try to simulate the behavior of a complex system over time. The simulation is broken down into shorter time steps on which the model is able to predict the outcome within reasonable accuracy. Such simulations tend to be computationally intensive and may run for hours and even for several days. These codes are also often very large involving up to several hundred thousands of code lines. If an error should occur in such a program, the task of locating it might be very difficult and time-consuming.

The development of large numerical codes is usually carried out in an incremental fashion and over a long period of time. Hence it is not unlikely that if parts of the code are changed and an error occurs one would have two versions of the program, one giving the correct answer and a new version giving the wrong one. In this case, it could be argued that the programmer has sufficient knowledge of what has been changed in the program in order to locate the error. But adding new features and capabilities to a program might initiate an error in the old parts of the code that have already been tested. Moreover, if the error only occurs for specific data sets the code might have continued developing for some time before one realizes that it contains an error.

We present a framework called *comparative debugging* that automates this particular debugging situation. In this framework, the programmer runs the old and the new versions of the program simultaneously, while monitoring the values of a set of selected variables. If the values of any variable differs between the two programs the programmer is automatically notified. This will direct the programmer to the first instance where an error occurs. It is also possible to trace the execution and ensure that the same statements are executed in the two programs.

We believe this to be a novel approach to automating and thus speeding up the debugging process that can easily be incorporated into existing debugging tools. In order to illustrate this we present a prototype tool called the *Wizard* that realizes the presented methods. This tool is developed using standard software components thus making it easily portable.

Apart from speeding up the debugging process the Wizard can also be used to verify that two programs that reach the same answer do so through the same set of computations.

The rest of the paper is organized as follows. In Section 2 we set up the the theoretical framework for comparative debugging. In Section 3 we point to some situations where this might be of use. The Wizard itself is presented in Section 4 before we conclude in Section 5.

2 Comparative Debugging

Based on the observation that most of the work in debugging is in locating where the error first occurs we suggest the following approach to automate the debugging process.

It is assumed that the user has two programs available: An old program A giving the correct answer and a new program B giving the incorrect answer. The task we wish to solve is to determine where the error first occurs in the new program. It is also conceivable that instead of an old program one has access through a file to the correct values of certain variables at different stages of the computation.

2.1 Comparative Breakpoints

We start by defining a comparative breakpoint $(E_A, l_A), (E_B, l_B)$. Here E_A is an expression involving variables from program A and l_A is a line number in A defining where E_A is to be evaluated. The variables used in E_A must be in scope in line l_A. The expression E_B and line number l_B is defined similarly for program B.

Each time program A executes line l_A the value of E_A is evaluated using the current values of the variables in A. The value of E_A is then stored in a queue. Similarly when program B executes line l_B the value of E_B is evaluated and stored in a separate queue. Whenever both queues are non-empty the first elements of the queues are compared for equality and deleted. Thus the value of E_A and E_B at the k'th time lines l_A and l_B are executed will be compared. This is true even if the programs should execute asynchronously.

We define two types of comparative breakpoints: blocking and non-blocking. When a program encounters a blocking breakpoint the program halts after storing its value to the queue. When both programs have stored their associated values and the comparison has been performed the programs are notified of the outcome. If $E_A = E_B$ both programs continue execution, but if $E_A \neq E_B$ they halt. Figure 1 illustrates the data-flow for a blocking breakpoint.

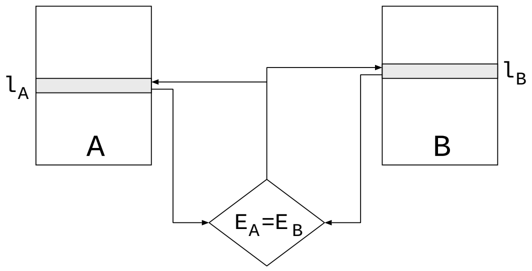

FIGURE 1. A blocking breakpoint

The rationale for this behavior is that if A and B are running inside two separate debuggers, control is passed back to the debuggers. So that in the event of a halt message the programmer can perform traditional debugging on the two programs before deciding on whether to terminate or continue the execution.

When a non-blocking breakpoint is encountered the program does not wait for the result of the comparison, but continues execution as soon as the associated value has been stored in the queue. If a mismatch is found when comparing the values in the queues an error message is output to the user. Consequently non-blocking breakpoints are mainly for detection of where an error occurs.

The main reason for including non-blocking breakpoints is speed of execution. A blocking breakpoint will cause the programs to synchronize at each breakpoint, slowing down the overall execution. Also if E_A and E_B are matrix values, or if breakpoints are encountered frequently, it may take a significant amount of time to transmit the necessary values and perform the comparison.

Apart from speeding up the execution, non-blocking breakpoints are slightly more flexible than the blocking ones. If A encounters a blocking breakpoint fewer times than B, this would cause A to hang indefinitely. Also as shown in Figure 2 if two blocking breakpoints have been defined such that $l_A < l'_A$ and $l'_B < l_B$, a deadlock might occur with program A waiting for input at l_A and program B waiting for input at l'_B. Both of these situations can be avoided by using non-blocking breakpoints.

Comparative breakpoints should preferably be used in such a manner

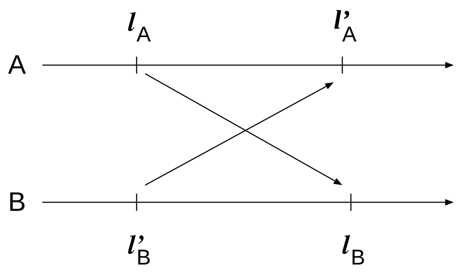

FIGURE 2. Blocking breakpoints may cause a deadlock

that deadlocks are avoided. It is, however, the responsibility of the user to ensure this.

In Section 3 we present some examples to motivate where comparative breakpoints can be of use.

2.2 Execution Tracing

The idea of comparative debugging can be extended to other areas in addition to comparing variables. We define a special type of comparative breakpoint called a *logical breakpoint* that can be used to trace and compare the execution of two programs.

A logical breakpoint is basically defined in the same way as the blocking and non-blocking breakpoints. The main difference is that when a program encounters a logical breakpoint only the identity of the breakpoint is used for comparison. So instead of associating an expression with each breakpoint we use an identifier i that is the same in both program A and B but is unique for each breakpoint. The formal definition of a logical breakpoint then becomes $(i, l_A), (i, l_B)$.

The only thing being compared during the execution of the two programs is the order in which the logical breakpoints are encountered. If program A encounters logical breakpoint i before breakpoint i' while program B encounters i' before i, an error will be reported.

To see how this might be used consider the following code segment:

```
while (sum < limit) {
      sum = sum + a[i];
      i = i + 1; }
return;
```

Assume that this particular segment of code is in both programs and that the user wants to know if it is being executed the same number of times in the two programs. This can be done by adding two logical breakpoints to the codes, the first attaching the two summation lines to each other and the second attaching the `return` statements to each other. Then an error will be reported if the programs execute the loop a different number of times. If a blocking logical breakpoint has been used, the program that executes the loop the fewest times will halt on the `return` statement, while the other program will halt on the summation statement. If non-blocking logical breakpoints are used an error will be reported but the programs will continue their execution. Note that in the above example, it is not sufficient to add a comparative breakpoint that compares the values i in the two programs. This is because the initial values of i might differ between the two programs.

Logical breakpoints should only be used where the execution flow of a program can branch. The obvious place being inside a loop or in an `if` statement. Other settings might be a subroutine with multiple `return` statements or a `case` statement.

To ensure that any discrepancy is discovered, each possible branch of execution that one wants to trace must be tagged by a logical breakpoint. This means that in a loop one should tag both a statement inside the loop as well as the first line following the loop. Similarly in an `if` statement, both the `then` and the `else` statement should be tagged. If there is no `else` statement the first line of code following the `if` statement should be tagged.

If one avoids tagging a certain branch of execution one of the programs can escape through this and later return to execute the necessary logical breakpoint, thus avoiding the detection of an error. A basic block of code is a sequence of source lines such that if the first line is executed the program cannot branch until the last source line has been executed. Thus it is only necessary to add logical breakpoints to the first statement of a basic block in each program.

We note that in some cases it is possible to simulate a logical breakpoint by the use of comparative breakpoints. This can be done by adding comparative breakpoints that compares the logical expressions that govern the branching of the execution flow. This is, however, more cumbersome and not as flexible as using a logical breakpoint. In particular, this is true if one is comparing two code segments where the structure of the governing logical expressions are different.

2.3 Floating Point Comparisons

The exact value of a floating point value might vary depending on issues such as the order of the operands, the machine precision, the implementation of standard numerical functions, etc. For these reasons it is seldom meaningful to compare two floating point values to determine if their values are exactly equal. To circumvent this problem we introduce a user specified value ϵ which is the largest tolerated difference when comparing two floating point values before an error is reported.

3 Examples of Use

In this section, we give a few examples where comparative breakpoints might aid the programmer both in verification of code and in the debugging process. In general, it is possible to use comparative debugging in any setting where a secondary program can generate the expected answer of a code segment.

3.1 Matrix Multiplication

The core of many block algorithms used in linear systems solvers, is a matrix multiplication routine. The time complexity of calculating the matrix product $C = AB$, where A, B, and C are $n \times n$ matrices, using traditional matrix multiplication, is $O(n^3)$. Strassen's matrix multiplication algorithm is able to perform this calculation in time $O(n^{2.807})$ [GL89, Str69]. Thus one might be able to speed up the code by using Strassen's algorithm instead of the traditional algorithm. Both IBM and Cray support routines for fast matrix multiplications using Strassen's algorithm. See also [Bai88, BMSV92] for examples of implementations of Strassen's algorithm. However, the error bound given by Strassen's algorithm is weaker than that of the traditional algorithm [Bre60, Hig90]. Therefore the two approaches might give different answers. If one suspects that this is the case one can use comparative breakpoints to determine if and when the result of Strassen's algorithm differs significantly from that of the ordinary algorithm.

3.2 Order of Execution (Sequential - Parallel)

For our second example we consider parallel computing. It is well known that the order of the operands in a computation can have significant impact upon the end result. See [Esp95] for a discussion of floating-point summations. A particularly computational intensive segment of code might be rewritten to run in parallel. If the end results of the sequential and parallel codes differ, one might wonder if this is the result of a coding error or of performing the computations in a different order. In this case, comparative debugging can be used to determine the magnitude and the first instance where the codes produce different results.

3.3 Different Input

If the input data to a program has been slightly perturbed the program might perform other actions then what it did previously. The user might then want to catch the first instance where the program behaves differently on the two datasets. This can easily be accomplished with the use of logical breakpoints.

3.4 Cross-platform Debugging

Finally, we note that if an error occurs only on a particular computer platform it is possible to use comparative debugging while running the same program simultaneously on two different computers and performing the communication between the two programs through the net.

4 The Wizard

To illustrate that the concept of comparative debugging is feasible and useful, we have implemented the features described in Section 2 in a debugging tool called the *Wizard*. The Wizard lets the user run two programs simultaneously, each running inside a debugger extended with comparative breakpoints.

4.1 Implementation

The Wizard is comprised of a monitor program and two debuggers. These are again merged inside a graphical user interface. The two debuggers are slightly modified version of the GNU debugger *gdb* [SP94]. The *gdb* debugger has been extended with comparative breakpoints as described in Section 2. These breakpoints are constructed in a similar way as the ordinary breakpoints only that the required expression has to be evaluated and transmitted to the monitor program. The comparative breakpoints use *gdb*'s own internal representation of variables. With this implementation it is possible to use all the ordinary debugging features included in *gdb*. It is also possible to make the evaluation of a comparative breakpoint conditional.

The monitor program is implemented in C and performs the actual comparisons of data transmitted from the two debuggers. The communication between the two debuggers and the monitor program is done using the PVM message-passing system [GBD+93]. The programs are assembled inside a user interface implemented using TCL/TK [Ous93, Wel94] and Expect [Lib94]. By using software that is available on most Unix platforms there is no major problem in porting the tool to new systems. Figure 3 illustrates the design of the Wizard.

4.2 The User Interface

When the Wizard is started, the user first specifies on which computers the programs are to run. The two programs to be executed are then loaded into the debuggers. Figure 4 shows the main view of the program.

There are four menus at the top of the screen. Their functions are as follows:

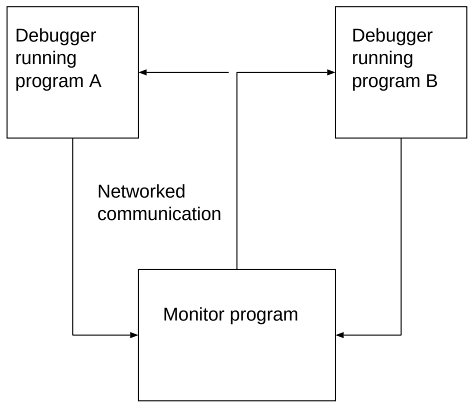

FIGURE 3. The design of the Wizard

- `File`, This menu is used to specify the executable programs to run in the debuggers.
- `Pvm`, This menu lets the user specify on which computers the programs are to be run. It is also possible to configure PVM through this menu.
- `Views`, This allows for pop-up windows containing graphical interfaces to a number of *gdb* commands.
- `Help`, Displays various help messages.

There are two windows listing the source codes of the two programs. Below these are two interactive windows used for input to the programs and for displaying messages from *gdb*. Below these windows are five push-buttons. The effect of these are as follows:

- `continue`, After a mismatch has been found at a blocking breakpoint both *gdb* processes halt. With this button the user can continue execution of the two programs.
- `delete`, This button deletes a comparative breakpoint (from both programs). One may thus be assured that both programs contain the same number of comparative breakpoints.
- `run`, This button starts the execution of the two programs.
- `blocking`, This button lets the user specify a blocking comparative breakpoint. The user gives the line numbers of the breakpoint and

specifies if this is a logical breakpoint or not. If not the user gives the expressions to be evaluated in each of the two programs.

- `non-blocking`, This button is similar to the "blocking" button only that a non-blocking comparative breakpoint is specified.

At the bottom of the user interface is a window displaying the output of the monitor program. Whenever a comparison of the values from a comparative breakpoint results in a discrepancy an error message is printed in the bottom-most window specifying the line-numbers of the breakpoint and the values that caused the mismatch.

Below the top-most menus there are three push-buttons, the left-most containing an image of a wizard and the two others displaying "GDB1" and "GDB2". The "GDB1" button is used only to view the first of the two programs. The user interface then changes so that the second program is no longer shown and a number of new push-buttons appear. These buttons display standard debugging commands for *gdb*. In this setting, the Wizard functions like a standard *gdb* debugger. Similarly "GDB2" displays only the second program, while the wizard button resets the display to that of Figure 4.

Since the Wizard has been implemented using explicit message passing it is possible to execute the two programs to be debugged on different computers. As mentioned in Section 3 this is advantageous if the error only occurs on a particular computer platform or if the memory requirements of the program are such that two versions cannot be executed simultaneously on the same computer.

5 Conclusion

We have introduced the notion of comparative debugging for automating the debugging process on large numerical codes. Comparative breakpoints were implemented in the Wizard thus showing the feasibility of this idea.

Tests performed so far indicate that the Wizard can be used to detect a number of errors. It can also be used to compare and verify the results of a segment of code continuously through a computation. Although this work is motivated from debugging numerical codes, the described methods can be applied to any debugging situation where it is possible to generate the expected answer to be produced by a segment of code.

The main obstacle to using comparative debugging is that one must have access to two very similar programs. Although this might not be a common situation for most programmers, we believe that when this does happen, using a tool like the Wizard will be worthwhile.

One of the main objectives of this study has been to show that comparative debugging can be incorporated in existing debugging tools. We believe that this has been achieved. Several debuggers have the possibility to control multiple threads or programs from one debugging session. Thus it should be fairly easy to extend these with comparative breakpoints.

A similar effort to the one described here has been presented by Abramson et. al [AFMR95, SA]. They describe a software system that can be attached to and control two existing debuggers. In this way they supply the user with comparative breakpoints similar to those described in Section 2.1. Their approach to comparative debugging requires the use of an software system in addition to the debuggers. Thus to run their system on a new platform requires porting the software system while the approach presented here uses a debugger that already exists on several platforms.

There are several extensions to the current work that would be useful in a future more complete tool. We mention some of these here:

- The current version of the program can only be used to debug codes written in C. We are currently working on extending the program to also handle Fortran programs.
- As of now the program is restricted to only compare scalars. One should allow for sending of whole arrays and array-segments.
- A comparative breakpoint inside a tight loop will generate a large number of data packages to be sent to the monitor program. This overhead will slow down the Wizard. One way this could be remedied is by specifying that data should only to be sent every n'th time a breakpoint is executed. The extra data could then either be packed into larger messages or discarded.
- Currently we do not allow for running a program against data stored in a file.
- The tolerance ϵ used in comparisons is the same for every breakpoint. It might be useful to have the option to specify an individual tolerance for each breakpoint.

It is our intention to make the Wizard available to other users. There is currently an extended web-presentation available on the Internet [A w].

6 References

[A w] Automating the debugging of large numerical codes. http://www.ii.uib.no fredrikm/fredrik/debug/.

[AFMR95] D. Abramson, I. Foster, J. Michalakes, and Sosic R. Relative debugging and its application to the development of large numerical models. In *Proceedings of IEEE Supercomputing 1995*, December 1995.

[Agr91] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs.* PhD thesis, Purdue University, West Lafayette, IN, 1991.

[Bai88] D. H. Bailey. Extra high speed matrix multiplication on the Cray-2. *SIAM J. Sci. Statist. Comput.*, (9):603–607, 1988.

[BMSV92] P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic. Efficient matrix multiplication on simd computers. *SIAM J. Matrix Anal. Appl.*, 13(1):386–401, 1992.

[Boe81] B. W. Boehm. *Software engineering economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[Bre60] R. P. Brent. Algorithms for matrix multiplication. Technical Report CS 157, Computer Science Department, Stanford University, Stanford, CA, 1960.

[Esp95] Terje Espelid. On floating-point summation. *SIAM Rev.*, 37(4):603–607, 1995.

[GBD+93] G. A. Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and V. S. Sundaram. *PVM 3 user's guide and reference manual.* Oak Ridge National Laboratory, 1993.

[GL89] G. H. Golub and C. F. Van Loan. *Matrix Computations.* North Oxford Academic, 2 edition, 1989.

[Hig90] N. J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Trans. Math. Software*, 16:352–368, 1990.

[Lib94] Don Libes. *Exploring Expect.* O'Reilly, 1994.

[Mye79] G. J. Myer. *The Art of Software Testing.* Wiley-Inter-Science, New York, 1979.

[Ous93] John K. Ousterhout. *TCL and the TK Toolkit.* Addison-Wesley, 1993.

[Pan93] Hsin Pan. *Software Debugging with Dynamic Instrumentation and Testbased Knowledge.* PhD thesis, Purdue University, West Lafayette, IN, 1993.

[SA] R. Sosic and D. A. Abramson. Guard: A relative debugger. To appear in Software Practice and Experience.

[SP94] Richard M. Stallman and Rolnd H. Pesch. *Debugging with GDB, The GNU source-level debugger*, 4.12 edition, 1994.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numer. Math*, 13:354–356, 1969.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Numer. Math.*, 7(25):446–452, 1982.

[Wel94] Brent Welch. *Practical Programming in TCL and TK.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Zel78] Marvin V. Zelkowitz. Perspectives on software engineering. *Numer. Math.*, 2(10):197–216, 1978.

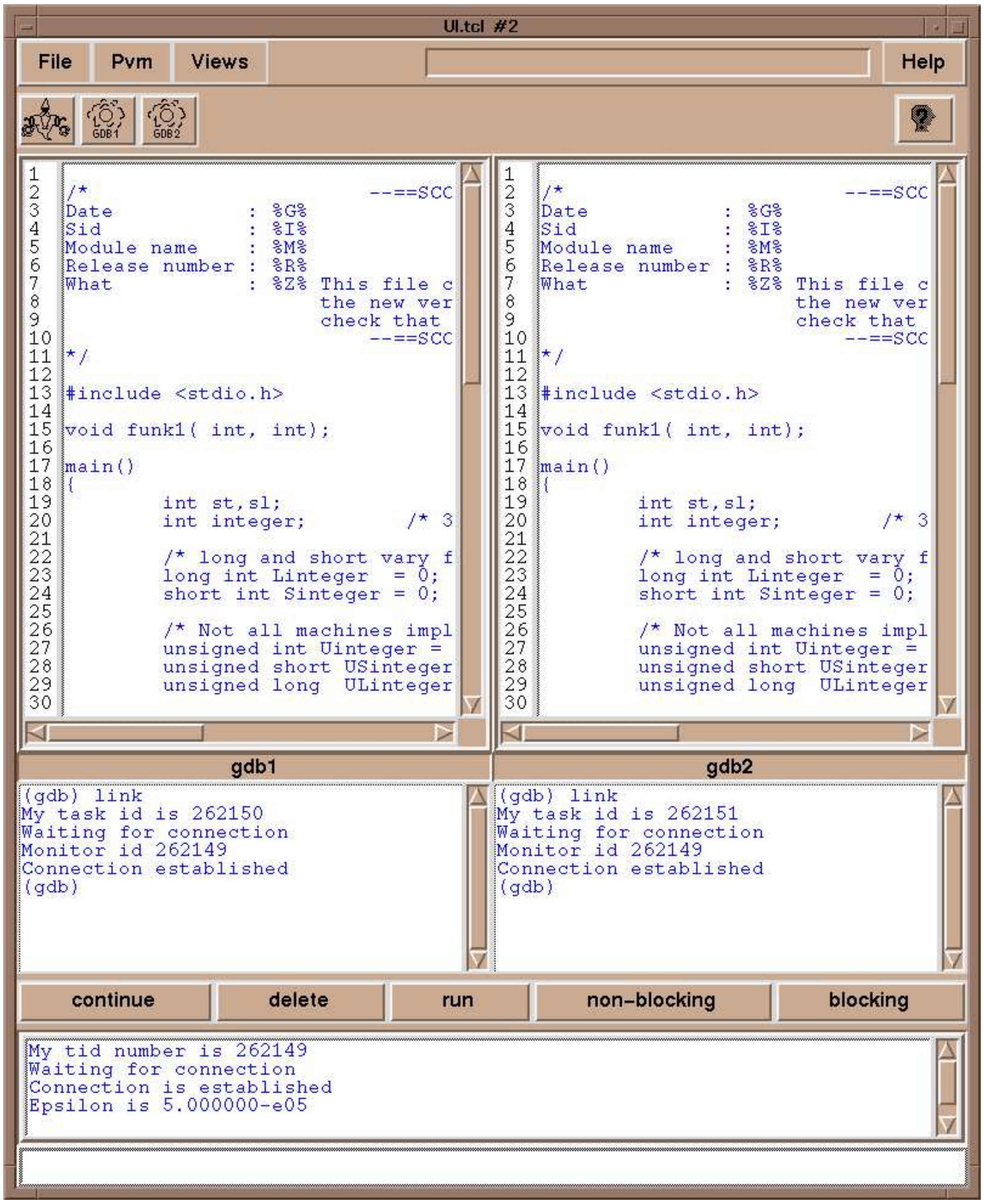

FIGURE 4. Main view of the Wizard