

A Debugging Tool for Software Evolution

D. Abramson
R. Susic

School of Computing and Information Technology
Griffith University
Kessels Rd, Brisbane,
Queensland, 4111
{davida, susic}@cit.gu.edu.au
Phone: +61-7-875 5049
Fax: +61-7-875 5051

ABSTRACT

This paper describes a tool for debugging programs which have been developed using evolutionary software techniques. The tool enhances the traditional debugging approach by automating the comparison of data structures between two running programs. Using this technique, it is possible to use early versions of a program which are known to operate correctly to generate values for comparison with the new program under development. The tool allows the reference code and the program being developed to execute on different computer systems by using open distributed systems techniques. A simple visualisation system allows the user to view the differences in data structures. By using the data flow of the code, it is possible to locate faulty sections of code rapidly. A small case study of finding an error in a scientific code is given.

1. INTRODUCTION

The high cost of software development, in combination with advances in software engineering, has caused the emergence of a software development methodology based on *evolution*. In this methodology, new programs are built from existing ones, utilising sections of code from old programs which perform functions required in the new application. The methodology is most commonly applied when the functionality of a program is expanded incrementally, and also when code is *ported* from one system to another. In the latter case, the old code is moved to a new platform with as little modification as possible. However, this simple minded approach often fails. For example, it may be necessary to modify sections of code which are system dependent, and in some circumstances it may even be necessary to rewrite the entire program in another programming language. Further, subtle differences in the semantics of programming languages and operating systems means that the code may behave differently on two systems. Because of these practical considerations, it is desirable

that software tools are available to simplify the process as much as possible.

CASE environments incorporating tools such as source code control systems like SCCS [1] and RCS [2]; program dependency tools like Make [1]; and machine independent libraries can be effective in assisting the development process. Examples of the latter include graphical user interfaces like X [3], which are available on a range of platforms, and thus provide a high level of infrastructure on different systems. In combination with standard programming languages, these tools are quite effective and can increase productivity significantly.

However, debuggers such as *dbx* [4, 5, 6] and *gdb* [7], and others [8, 9, 10, 11, 12] do not emphasise the debugging and testing of applications developed with evolutionary engineering techniques. Debuggers of this type allow the user to manipulate the new program through process control commands, and to examine and modify the state of the program. In debugging the program, the user must determine possible modes of failure, and then stop the execution at key points to examine the state. In determining whether the state is correct or not, the user must be able to *predict* the state values. The prediction is usually based on a detailed knowledge of the operation of the code. This can be extremely difficult for complex codes, especially if the person performing the debugging is not the author of the original program.

In this paper we discuss a new debugging tool which can be used to test programs which are developed by *evolution*. Our tool incorporates conventional debugger commands as well as a new set of commands which facilitate comparison of the new program with an existing version. The paper begins with a discussion of the current techniques used to test evolutionary

programs. It then describes *Guard*¹ (Griffith University Relative Debugger), a new debugger developed at Griffith University. Finally, we give an example of the use of *Guard* through a case study.

2. HOW DO WE TEST AND DEBUG EVOLVING PROGRAMS?

A common life cycle for a program developed using evolutionary engineering techniques is shown in Figure 1. It often begins as a small program for testing some basic functionality and design. It may be augmented and incrementally expanded into a large software system, and some of the core algorithms may even be altered. These types of modifications can be attributed to changes in the functionality or operation of the code. The program may be converted for execution on different hardware and software platforms, and may even be re-written in another language to allow it to take advantage of new hardware. These types of modification can be classified as migratory ones. Regardless of the cause of the changes, at each of these stages the programmer must determine whether the program is still operating correctly, and if not, must determine which of the alterations caused the new behaviour.

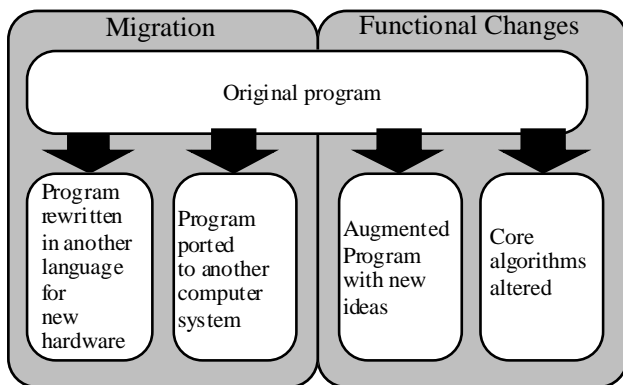


Figure 1 - Typical life cycle of software system

Currently, there are very few tools which assist porting and debugging evolving programs across different hardware and software platforms. Traditional debuggers have a number of severe limitations which significantly reduce their applicability. The most serious limitation is that they are incapable of dealing with several programs running at the same time and possibly on different computers. They have no facilities for comparing the output and internal state of different programs, which must be done by tedious and error prone manual methods.

The most common technique for testing and debugging evolved programs is to use the data flow of the code to locate the point at which the data structures in the new program diverge from those in the existing one. Thus, the existing code acts as a *reference* version by defining a set of expectations. In this way, the user typically works back to determine the last point at which the data was correct by comparing data structures with the reference version. Once this point has been established, most errors can be traced quickly to a small section of faulty code, and the error can be corrected. Debugging real programs using this technique and currently available tools can be tedious and error prone. Typically, output statements are placed in both the reference and the debugged code, and the values are either compared by visual inspection, or by a file comparison program [13]. If the two programs execute on different computer systems then the data must be transferred before it can be compared.

Because of these shortcomings, we have developed a new debugging tool, and this will be described in the next section.

3. Guard: A RELATIVE DEBUGGER

3.1 What is Guard?

Guard is a distributed debugger which operates in an open heterogenous computing environment [14, 15]. *Guard* provides the user with functionality to control more than one program and to make assertions about the correctness of a new program with reference to an existing one. Consequently, *Guard* supports the evolution of programs because it acknowledges the existence of working versions of the code. Figure 2 shows the way that *Guard* interacts with both the reference code and the code being debugged.

Because *Guard* works in a distributed environment, it is possible to execute the reference code, the program being debugged and *Guard* itself on three different computer systems, as shown in Figure 2. *Guard* communicates with the programs it is controlling via a network and makes use of special network support libraries linked with the applications.

Guard relies on the premise that differences between the data structures in the reference code and the one being debugged can be used to detect faults. This assumes that the two programs utilise comparable data structures, or at least provide a set of conversion routines which makes them appear equivalent. *Guard* makes no assumptions about the control flow of the two programs, which may be different. It requires the user to determine key points in the two programs at which various data structures should be equivalent.

¹ Patent Pending

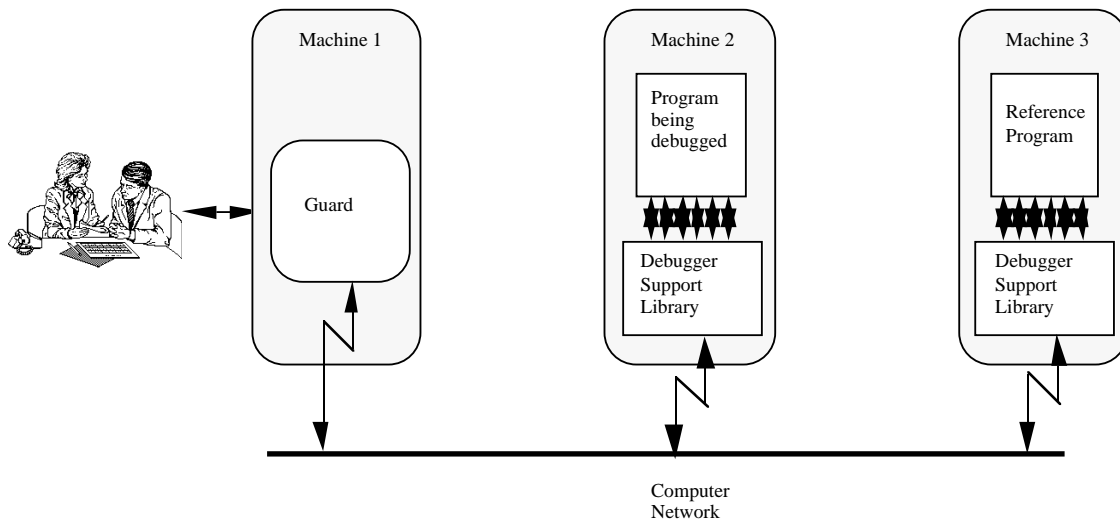


Figure 2 - *Guard* - A relative debugger

the faulty region, such as a binary search, can be very effective.

3.2 Network independent debugging

Guard makes use of open distributed processing techniques to allow the reference code and the debugged code to execute concurrently on different computer systems. This means that when a code is being ported to another system, the reference code operates in an environment which is known to produce the correct results. Consequently, the user is able to concentrate on the causes of deviation of the debug code.

Network location information is restricted to the *Guard* command `invoke`, which starts executing a process. In this command a local name is bound to the process for use in future commands. After the `invoke` command has been issued, all other *Guard* commands use the logical name when referring to the process.

3.3 Using *Guard*

Guard relies on the user to make a number of assertions which compare data structures in the debugged code and the reference version. These assertions make it possible to detect faulty code because they indicate where (and when) the data structures deviate from those in the reference code.

The choice of data structures and test points must be determined by the user based on some knowledge of the application. It is not necessary to test all data structures, but only those which will help uncover the source of the error. Likewise, it is not necessary to test the structures after each executable statement. A search which refines

Guard can be used in two modes: one in which the assertions are specified *declaratively*, and the other using *procedural* techniques. The procedural scheme relies on the user manually placing breakpoints in both the reference code and the debugged code. These are planted at key test points. Both programs are then executed until they reach these breakpoints, after which the user tests the equality of arbitrary data structures using a `compare` statement. The following example shows the syntax of the `compare` statement by comparing the values of variable `test` in the reference code and the debug code. In this example, `reference` and `debug` are names which are bound to the two processes under examination by the `invoke` command.

```
compare reference::test = debug::test
```

If the two data structures are not equivalent, then the nature of the error is reported. If the variables are simple types like integer or real, then it is possible to report the two values. If they are compound structures like arrays, then the location of the difference in the two structures must be reported as well. In a future section we will describe the exact method of reporting differences for array type structures. The user can then resume execution of the two programs using a `continue` statement. It is possible to compare a number of data structures after the breakpoints have been reached, and also new breakpoints can be set to further refine the erroneous code. This process closely resembles the way a conventional debugger is used. However, it allows the user to control two processes concurrently and compare data structures. This manual scheme can become unwieldy on large programs because there are two processes to control.

Further, it is not well suited for using the debugger to automatically test whether a new version of program conforms to the previous one, because it requires a great deal of user interaction. Consequently, we developed another, automatic, mode of operation called declarative assertions.

Declarative assertions allow the user to bind a set of breakpoints to comparison operations at any time prior to or during execution of the code. In this way, the comparisons are invoked automatically each time the breakpoints are reached. If the compare statements do not detect an error, then the code is automatically resumed without any user interaction. Declarative assertions provide a convenient mechanism for stating a set of conditions for correct execution, and thus are well suited for automatically testing a new program against previous versions. If the assertions do not cause any errors, then the code is known to conform to previous versions. Declarative assertions are also effective when an error is detected only after a number of iterations of a particular statement. Because the user is not involved until an error is detected, little user interaction is required to actually detect the erroneous lines. The following syntax is used to declare an assertion:

```
assert reference::variable1@line1 =
      debug::variable2@line2
```

where `reference` and `debug` are bound to the two processes as discussed previously, `variable1` and `variable2` are arbitrary variables in the programs, and `line1` and `line2` are source code line numbers. Later in the paper we will illustrate the use of declarative assertions for detecting errors in a large scientific modelling program.

3.4 Data Types

Like conventional debuggers, *Guard* needs to understand the types of variables in the programs it controls. On many systems (e.g. Unix) the type information is embedded in special symbol table information stored in the executable image. This can be automatically extracted by the debugger at the beginning of a debug session. In a conventional debugger, this information allows it to display variables in a way which is meaningful to the programmer. In *Guard*, the information is also required so that it knows how to compare variables. For example, variables must be type compatible before they can be compared. Further, *Guard* needs to understand the structure of the variables during the comparison operation so that it can traverse the basic elements. Other complications arise because the reference code may execute on a system with different attributes from the code being tested. For example, byte

ordering may differ and the two programs may even be written in different languages. Consequently, *Guard* must map types from one system onto another.

Guard currently handles variables of base types integer, real, character and multi dimensional arrays of these base types. The base types are compared for equality. Arrays are compared by traversing each of the indices in order, and differences are reported together with their position information. This allows *Guard* to compare arrays in programming languages which use different ordering, such as Fortran and C. *Guard* allows also comparison of sub-arrays .

These base types have been sufficient to demonstrate the effectiveness of *Guard* in debugging a number of large scientific programs. In future versions of *Guard* we will add support for data types such as records and dynamic structures like linked lists. The latter requires sophisticated procedures for mapping the data structures into an intermediate form prior to comparison and for interpretation of this intermediate form.

3.5 Tolerances

A program may not be in error simply because its variables do not exactly match those of a reference version. The most obvious example of this is when the two systems use different floating point representations or libraries. In this case numbers may be deemed to be equivalent if they are within a predefined tolerance of each other. Accordingly, *Guard* includes a user controlled tolerance threshold, below which numbers are considered equivalent. Further, individual assertions may specify their own threshold rather than using the global one.

To allow the user to explore the way in which two variables differ it is possible to vary the threshold through a slider control. This allows the user to view the maximum regions without explicitly requesting the values.

3.6 Displaying the results

If two scalar variables differ then it is possible to display the two values and report the difference. However, when complex data structures differ it is difficult to interpret the cause of the error by viewing the numeric values alone, particularly if they are floating point numbers. Consequently, we have developed a simple visualisation system for *Guard* which uses a pixel map to show when corresponding values of array type data structures are not the same. *Guard* also reports the maximum and average differences between arrays as numeric values.

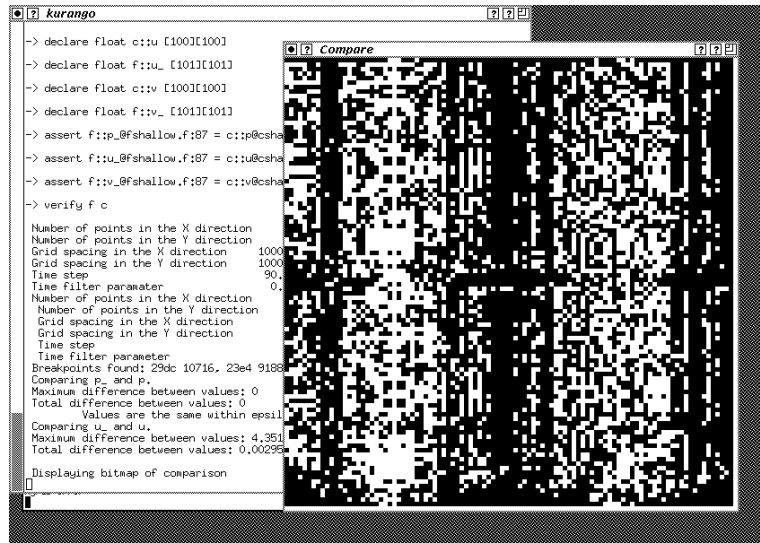


Figure 3 - Numeric instability causing errors

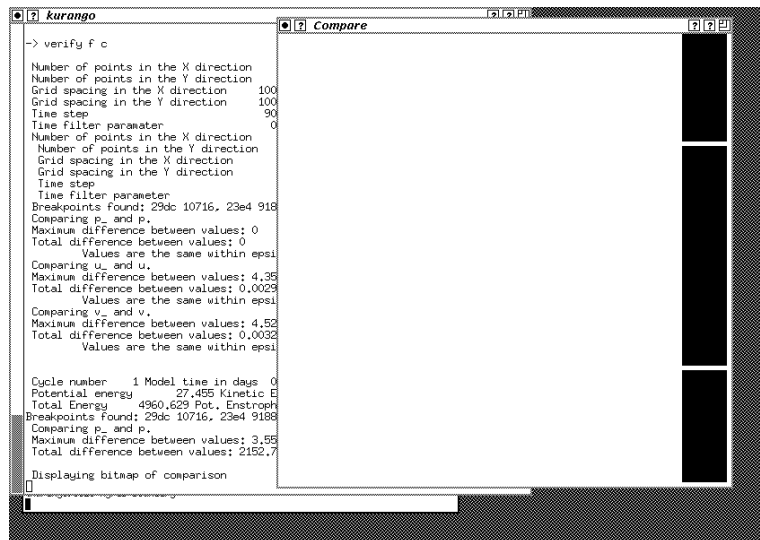


Figure 4 - Incorrect loop bound

The most intuitive display is formed when two dimensional arrays are visualised. In this case, a two dimensional pixel map is created in which each pixel corresponds to one array element. In this way errors which are caused by incorrect loop bounds and strides are obvious because regular patterns are created. *Guard* currently maps all other shaped arrays onto two dimensional ones by either expanding the one dimensional arrays or merging higher dimensions. It is our intention to experiment with three dimensional visualisation techniques to show three dimensional difference patterns, as well as colour for higher order

arrays. We have not yet considered suitable display methods for dynamic data structures.

Figures 3 and 4 show some sample visualisations which were produced by *Guard* when it was used for testing a new version of the Shallow Water Equations [16]. The original code was written in FORTRAN and the new version was written in C and ran on a different computer system. Figure 3 shows the effect of floating point differences between two of the two dimensional data structures used to hold the *pressure* of the wave. In both visualisations, a black pixel means that the data is different at the specified row and column of the arrays.

From Figure 3 it can be seen that the two data structures are drifting apart due to numeric errors because many of the values are incorrect, but the maximum difference is quite small (4.3×10^{-06}). Figure 4 shows the effect of a

wrong loop bound in the new code. The section of the array which has not been calculated can be clearly seen as a band of black pixels covering the missing columns on the right.

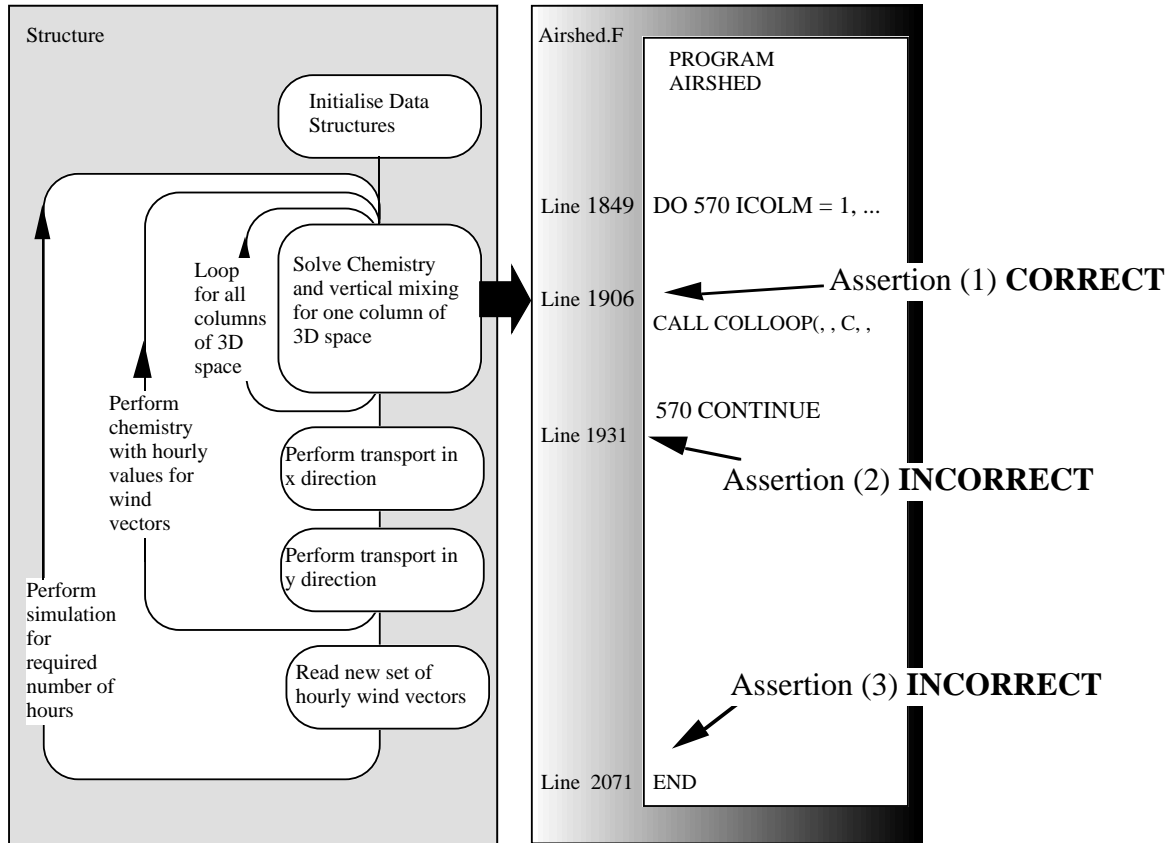


Figure 5 - Overall structure of code plus source in Airshed.F

4. A CASE STUDY

In this section we describe an application of *Guard* for finding a subtle error in a scientific code. The program, a photo chemical pollution code, models the chemical processes which occur during smog formation [17]. It has been used as part of a number of real world studies involved with formulating pollution control strategies. Table 1 shows the characteristics of the software, which are typical of many other scientific modelling programs.

One of the key data structures in the program is an array, named C, which holds the *concentrations* of all chemical species for each of the cells in three dimensional space. The array is conceptually indexed by two co-ordinate indices (column number in 2D space and vertical level) and a chemical species number. Whilst this would normally require a three dimensional array, it is actually represented as a one dimensional array. This assists with the vectorisation of the program, and also with dynamic

memory allocation. However, it also makes the code harder to understand and debug.

Language	Fortran
Lines of code	15,000
Number of source modules	15

Table 1 - Characteristics of Smog Model

In this study we show how *Guard* was used to track an error in the numeric integration code. The problem was detected after the program failed to generate correct results for a particular simulation. A reference version of the code was established and used for the comparison of key data structures. Whilst both versions were run on the same computer, it would have been possible to execute them on different networked systems.

Figure 5 shows basic computational structure of the program at the outer level. After initialising the key data structures the program enters a loop in which the concentrations of each of the chemical species are calculated at discrete time steps. The program works on sections of the concentration vector (C) corresponding to each column of the 3 dimensional space. Once the new concentrations have been computed horizontal transport is performed in 2 directions. The program uses a set of hourly wind vector values a number of times before reading in a new set of vectors. These operations are mostly performed in the source file `airshed.F`.

The assertions shown in Figure 6 were used to determine the point at which the concentration vector (C) became corrupt. As discussed earlier, the assertions were placed at strategic places in the code to try and locate where the C vector became corrupt. These assertions detect that the vector C was incorrect at lines 2071 and 1931 but still correct at line 1906. Consequently, the error must be contained in the routine `COLLOOP`.

```

assert  old::C@airshed.F:1906 =
        new::C@airshed.F:1906
Assertion(1)
assert  old::C@airshed.F:1931 =
        new::C@airshed.F:1931
Assertion(2)
assert  old::C@airshed.F:2071 =
        new::C@airshed.F:2071
Assertion(3)

```

Figure 6 - Assertions relating to `Airshed.F`

Figure 7 shows the structure and source of the routine `COLLOOP`. The appropriate cells from C are copied to a temporary concentration vector called CNT. The new concentrations for the species are calculated by solving the ordinary differential equations which govern the rates of production of each chemical species. Then the concentrations are adjusted to take account of vertical mixing in the column. These two operations are mostly performed in `colloop.F`. The assertions shown in Figure 8 were used to conclude that the values in CNT were correct at line 305 and incorrect at line 321. Consequently, the error must be contained in the routine `INTEGR2`.

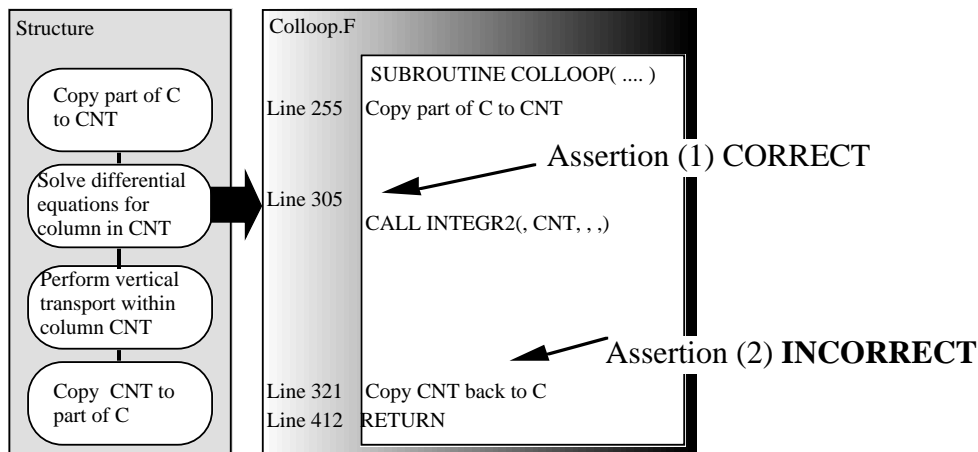


Figure 7 - Structure and code of `COLLOOP` in `Colloop.F`

```

assert  old::CNT@colloop.F:305 =
        new::CNT@colloop.F:305
Assertion(1)
assert  old::CNT@colloop.F:321 =
        new::CNT@colloop.F:321
Assertion(2)

```

Figure 8 - Assertions relating to `Colloop.F`

Figure 9 shows the structure and code of the routine `INTEGR2`, which performs a numeric integration. This makes use of a number of working vectors (such as C3). Figure 10 shows the assertions related to `INTEGR2`. These assertions determine that C3 was incorrect at 711 and 595, but the switch variable IS was correct at line 587. From the information gathered by these assertions the error was found at line 587 of `numerics.F` as shown in Figure 11.

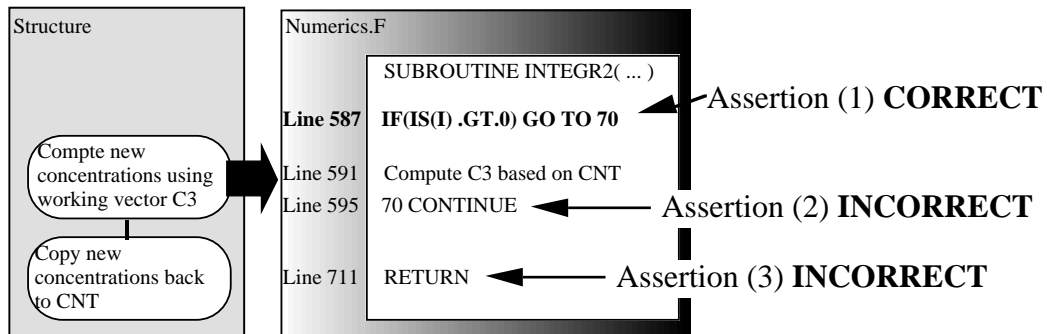


Figure 9 - Structure and code of `INTEGR2` in `Numerics.F`

```

assert old::IS@numerics.F:587 =
new::IS@numerics.F:587
Assertion(1)
assert old::C3@numerics.F:595 =
new::C3@numerics.F:595
Assertion(2)
assert old::C3@numerics.F:711 =
new::C3@numerics.F:711
Assertion(3)

```

Figure 10- Assertions relating to `Numerics.F`

<pre>IF (IS(I) .GT.0) GO TO 70</pre> <p style="text-align: center;">Correct</p>
<pre>IF (IS(I) .LT.0) GO TO 70</pre> <p style="text-align: center;">Incorrect</p>

Figure 11 - Correct and incorrect source in `numerics.F`

This error caused only some of the array elements to be incorrect, and would have been extremely difficult to trace using a conventional debugger. The difference visualisations allowed the error to be detected very quickly using a simple search in combination with the data flow in the code. With these assertions in place the incorrect code was located after some 2-3 minutes of execution time. One of the main attractions of *Guard* was that it was not necessary to alter the code during the debugging phase, and thus new assertions could be developed and refined without the need to recompile the code.

5. CONCLUSION AND FUTURE DIRECTIONS

In this paper we have described a new tool which supports the debugging and testing of programs

developed with evolutionary software engineering techniques. The tool recognises previous versions of a program and allows comparison of data structures between new and old versions. Because it operates in a distributed heterogenous environment it is ideal for use in program porting because the original implementation can act as a reference site. Through a simple case study, we have illustrated the power of the system.

Guard is also useful for automatically testing new versions of programs against existing ones. The declarative assertion makes it possible to specify a number of assertions about key data structures before any changes are made to the code, and the program can be executed under the control of the debugger to verify these that these assertions are met. In this paper we have ignored a number of high level features of *Guard*, such as partial assertion lists and generation of trace files. These two features are important because they simplify this automatic process. More details can be found elsewhere [15].

Current research involves expanding *Guard* to support the testing of parallel programs, supporting more data types and increasing the range of data structure visualisations which are possible. The tool has been ported to SUN, Next and Silicon Graphics machines, and a port to IBM RS6000 is nearly complete. We are planning a parallel version for the IBM SP2 during 1995.

Acknowledgments

This work has been sponsored by the Australian Research Council. The authors wish to acknowledge the work of Lisa Bell, who has performed most of the programming necessary to implement *Guard*. Thanks go to Simon Wail from IBM for proof reading a draft of this paper.

References

- [1] 'Programming Utilities and Libraries', Sun Release 4.1, Sun Microsystems, 1988.

- [2] Walter F. Tichy, "RCS-A System for Version Control", *Software-Practice & Experience* 15, 7 (July 1985), 637-654.
- [3] A. Nye, *Xlib Programming Manual*, O'Reilly & Associates, Sebastopol, CA, 1989.
- [4] 'Debugging Tools Manual', Sun Release 4.1, Sun Microsystems, 1990.
- [5] E. Adams and S. S. Muchnick, 'Dbxtool: a window-based symbolic debugger for Sun Workstations', *Software-Practice & Experience*, 16, (7), 653-669 (1986).
- [6] M. A. Linton, 'The evolution of Dbx', *Proceedings of the Summer 1990, USENIX Conference*, June 1990, pp. 211-220.
- [7] R. M. Stallman, *GDB Manual*, Free Software Foundation, Cambridge, MA.
- [8] T. G. Moher. "PROVIDE: A process visualization and debugging environment", *IEEE Transactions on Software Engineering*, 14(6):849-857, June 1988.
- [9] R.A. Olsson, R. H. Crawford, and W. W. Ho. "A dataflow approach to event-based debugging", *Software-Practice and Experience*, 21(2):209-229, February, 1991.
- [10] N.Ramsey and D.R. Hanson. "A retargetable debugger"., *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 22-31. ACM, 1992.
- [11] E.Satterthwaite."Debugging tools for high level languages", *Software-Practice and Experience*, 2(3):197-217, July-September 1972.
- [12] D. Cheng and R. Hood, "A Portable Debugger for Parallel and Distributed Programs", *Proceedings of Supercomputing 94*, Dec 94.
- [13] N. Galbreath, W. Gropp, and D. Levine "Applications-driven parallel I/O", *Proceedings Supercomputing-93*, Portland, Oregon, pages 462-471. IEEE, 1993.
- [14] D.A. Abramson. and R. Sasic, "Relative Debugging: A new debugging paradigm", Griffith University Technical Report, School of Computing and Information Technology, Report # CIT-94-13.
- [15] Sasic, R. and Abramson, D. A. "Guard: A Relative Debugger", Griffith University Technical Report, School of Computing and Information Technology, Report # CIT-94-21.
- [16] D.A. Abramson, M. Dix, P. Whiting. "A Study of the Shallow Water Equations on Various Parallel Architectures", 14th Australian Computer Science Conference, Sydney, 1991.
- [17] McRae G.J., Russell A.G. and Harley R.A., 1992. CIT photochemical airshed model -users manuals, Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA.