

Relative Debugging using Multiple Program Versions

David Abramson and R. Sosič

School of Computing and Information Technology
Griffith University
Brisbane, QLD 4111
Australia
D.Abramson@cit.gu.edu.au

This talk will address software tools for debugging programs that have been re-engineered. Software re-engineering is a very broad field, and covers the following scenarios:

- A program may be augmented with some new features
- A program may be moved to a new computer or operating system
- A program may be moved to a different architectural model (i.e. parallel)
- A program may be re-written in another language
- A program may be specified using a dataflow or functional language and may be implemented using an imperative language.

In each of these scenarios multiple versions of the program exist, and the new version may fail and may need to be debugged. In each case a working version of the software is available for comparison with the new code. However, traditional debuggers are very poor at facilitating comparisons between executing software, and are designed to control only one program at a time. They are also poor at displaying large amounts of numeric data, as experienced in large scientific codes. Thus, tracking the cause of the failure may be extremely time consuming and error prone.

We have developed a tool which supports a new debugging paradigm called *relative debugging*, that is, a program is debugged relative to another working *reference* version of the code. The tool, Guard, provides powerful mechanisms for comparing data structures, even when they do not have identical forms in the two programs. By using state of the art visualisation programs we are able to illustrate where codes differ graphically.

Guard allows the two codes to execute on different platforms connected by the Internet. This means that it is possible to debug a program without having the correct version on the same machine as the one being tested. Guard uses open distributed processing techniques to allow transparent and concurrent execution of the two codes. These are supported by a multi-lingual, multi-vendor portable debugging package called Dynascope.

We have tested Guard on a number of real world scientific models, including a large meso-scale weather model consisting of about 40,000 lines of Fortran code. In this case one of the programs is a sequential, and the other is written for a parallel machine. The two codes are actually quite different in structure, and in some cases use different data representations for the same information. For example, the two programs used different ordering of array indexes. Also, the parallel version of the code included *ghost bands* around the data structure arrays to allow them to pass regions between processors using message passing primitives. Using Guard, it was possible to isolate an observed error in the output variables (each of which was a three dimensional (61 x 61 x 24) array of floating point numbers). By piping the Guard output into

IBM's Data Explorer software, it was possible to view the error in three dimensions as it grew on each iteration of the program.

We have also tested Guard by debugging a version of the Shallow Water equations written in C against a reference code written in Fortran. In this case the two programs used different control structures, and to a limited extent different data structures. The Fortran code was geared for a vectorizing compiler, whereas the C code was structured for parallel execution on a shared memory multi-processor. However, it was possible using Guard to establish points at which equivalence was expected, and this was used to detect bugs in the C code. For simple two dimensional arrays, Guard provides a basic visualisation facility, and this allowed the display of loop termination errors in the C code. This appeared as a regular pattern of black pixels (representing incorrect data) against a background of white pixels, indicating that entire columns of the array had not been computed. Guard understands the key differences between data representation in different languages, such as memory layout in arrays.

Relative debugging has some interesting applications in comparing execution states between many different programming paradigms. For example, it is conceptually possible to compare execution states of functional and imperative programs. This leads to the idea of building small, concise, functional specifications for programs and then comparing their behaviour dynamically with imperative versions of the algorithm.

Relative debugging also poses the possibility of establishing reference sites for popular software. When the code is ported to a new platform, the reference sites can be used to establish the correctness of the new version, across different platforms. This talk will discuss relative debugging, our prototype Guard and possible future work.