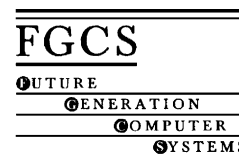




ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Future Generation Computer Systems 19 (2003) 665–678



www.elsevier.com/locate/future

Debugging scientific applications in the .NET Framework

David Abramson*, Greg Watson¹

School of Computer Science and Software Engineering, Monash University, Clayton, Vic. 3800, Australia

Abstract

The Microsoft .NET Framework represents a major advance over previous runtime environments available for Windows platforms and offers a number of architectural features that would be of value in scientific programs. However there are such major differences between .NET and legacy environments under both Windows and UNIX, that the effort of migrating software is substantial. Accordingly, software migration is unlikely to occur unless tools are developed for supporting this process. In this paper we discuss a ‘relative debugger’ called Guard which provides powerful support for debugging programs as they are ported from one environment or platform to another. We describe a prototype implementation developed for Microsoft’s Visual Studio.NET—a rich interactive environment that supports code development for the .NET Framework. The paper discusses the overall architecture of Guard under VS.NET and highlights some of the technical challenges that were encountered during its development. A simple case study is provided that demonstrates the effectiveness of relative debugging in locating subtle errors that occur when even a minor upgrade is attempted from one version of a language to another. For this example, we illustrate the use of relative debugging using a Visual Basic program that was ported from Visual Basic 6.0 to Visual Basic.NET.

© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Microsoft .NET Framework; Common Language Specification; Relative debugger

1. Introduction

The .NET Framework is a major initiative by Microsoft that provides a uniform multi-lingual platform for software development [17]. It is based on a Common Language Specification (CLS) that supports a wide range of programming languages and runtime environments. In addition, it integrates web services in a way that facilitates the development of flexible and

powerful distributed applications. Clearly this has applicability in the commercial domain of e-commerce and P2P networks which rely primarily on distributed applications.

An analysis of the features available in .NET suggests that the new architecture is as equally applicable to scientific computing as to commercial applications. In particular .NET provides efficient implementations of a wide range of programming languages, including FORTRAN [12], because it makes use of just-in-time compilation strategies. Further, the Visual Studio development environment is a rich platform for performing software engineering as it supports integrated code development, testing and debugging from the one tool.

Some of the more advanced features of .NET, such as Web Services, could also have interesting application in scientific code. For example it would be

* Corresponding author. Tel.: +61-03-9905-1183;
fax: +61-03-9905-5146; mobile: +61-0417-375-635.
E-mail addresses: davida@csse.monash.edu.au (D. Abramson),
gwatson@lanl.gov (G. Watson).
URLs: <http://www.csse.monash.edu.au/~davida>,
<http://www.acl.lanl.gov/cluster/people/gwatson>

¹ Present address: MS B287, Advanced Computing Lab, Los Alamos National Laboratory, Los Alamos, NM 87545, USA.

possible to source libraries dynamically from the Web in the same way that systems like NetSolve [7] and NEOS [9] provide scientific services remotely. This functionality could potentially offer dramatic productivity gains for scientists and engineers, because they can focus on the task at hand without the need to develop all of the support libraries.

Unfortunately the differences between .NET and other legacy software systems, such as WIN32 and even UNIX, are substantial and as a result there is a significant impediment to porting codes from one environment to another. Not only are the environments different functionally, but the libraries and machine architectures may differ as well. It is well established that different implementations of a programming language and its libraries can cause the same program to behave erroneously. Because of this the task of moving code from one environment to another can be error prone and expensive. Many of these applications may also be used in mission critical situations like nuclear safety, aircraft design or medicine, so the cost of incorrect software can potentially be enormous. Unless software tools are developed that specifically help users in migrating software to the .NET Framework, it is likely that most scientists will continue to use legacy platforms for their software development.

Traditional debuggers are not particularly helpful at finding errors introduced during the porting process even when they have been designed with scientific or distributed computing in mind [6,8,15,16,19,25]. This is because they generally require the programmer to have a good understanding of the way the program works and have a mental model of the contents of the various data structures during execution. In this paper we describe a debugging tool called Guard, which specifically supports the process of porting codes from one language, operating system or platform to another. Guard has been available under UNIX for some time now, and we have proven its applicability for assisting the porting of programs many times. We have recently implemented a version of Guard that is integrated into the Microsoft Visual Studio.NET development environment. Not only can the system be used to support porting from WIN32 to .NET, but we have even demonstrated the ability to support cross-platform debugging between a UNIX platform and a Windows platform. This has shown that the tool is not only useful for supporting software development on the one

platform, but can also support the porting of codes between Windows and UNIX.

The paper begins with a discussion of the Guard debugger, followed by a description of the .NET Framework. We then describe the architecture of Guard as implemented under Visual Studio.NET, and illustrate its effectiveness in locating programming errors in this environment.

2. Guard—a relative debugger

Relative debugging was first proposed by Abramson and Soscic in 1994. It is a powerful paradigm that enables a programmer to locate errors in programs by observing the divergence of key data structures as the programs are executing [1–5,20,24]. The technique of relative debugging allows the programmer to make comparisons of a *suspect* program against a *reference* code. It is particularly valuable when a program is ported to, or rewritten for, another language or computer platform. Relative debugging is effective because the user can concentrate on *where* two related codes are producing different results, rather than being concerned with the actual values in the data structures. Various case studies reporting the results of using relative debugging have been published [1–3,14,24], and these have demonstrated the efficiency and effectiveness of the technique. The concept of relative debugging is both language and machine independent. It allows a user to compare data structures without concern for the implementation, and thus attention can be focussed on the cause of the errors rather than implementation details.

To the user, a relative debugger appears as a traditional debugger, but also provides additional commands that allow data from different processes to be compared. The debugger is able to control more than one process at a time so that, once the processes are halted at breakpoints, data comparison can be performed. There are a number of methods of comparing data but the most powerful of these is facilitated by a user-supplied *declarative assertion*. Such an assertion consists of a combination of data structure names, process identifiers and breakpoint locations. Assertions are commands that are processed by the debugger before program execution commences and used to build an internal graph [5] which describes when the two

programs must pause, and which data structures are to be compared. In the following example:

```
assert $reference::Var1@1000
    = $suspect::Var2@2000
```

the `assert` statement compares data from `Var1` in `$reference` at line 1000 with `Var2` in `$suspect` at line 2000. A user can formulate as many assertions as necessary and can refine them after the programs have begun execution. This makes it possible to locate an error by placing new assertions iteratively until the suspect region of code is small enough to inspect manually. This process is incredibly efficient. Even if the programs contain millions of lines of code the debugging process refines the suspect region in a binary fashion so it only takes a small number of iterations to reduce the region to a few lines of code.

Our implementation of relative debugging is embodied in a tool called Guard. We have produced implementations of Guard for many varieties of UNIX, in particular Linux, Solaris and AIX. A parallel variant is available for debugging applications on shared memory machines, distributed memory machines and clusters. Currently this is supported with UNIX System V shared memory primitives, the MPICH library, as well as the experimental data parallel language ZPL [24].

The UNIX versions of Guard are controlled by a command line interface that is similar in appearance to debuggers like GDB [21]. In this environment an `assert` statement such as the one above is typed into the debug interpreter and must include the actual line numbers in the source as well as the correct spelling of the variables. As discussed later in the paper Guard is now integrated into the Microsoft Visual Studio environment and so is able to use the interactive nature of the user interface to make the process of defining assertions easier.

3. Success stories

Over the last few years we have used Guard to debug a number of scientific codes that have been migrated from one platform to another or from one language to another (or both). In one case study we used Guard to isolate some discrepancies that occurred when a

global climate model was ported from a vector architecture to a parallel machine [2]. This study illustrated that it is possible to locate subtle errors that are introduced when programs are parallelised. In this case both models were written in the same language, but the target architecture was so different that many changes were required in order to produce an efficient solution. Specifically, the mathematical formulation needed to be altered to reduce the amount of message passing in the parallel implementation, and other changes such as the order of the indexes on key array data structures needed to be made to account for an RISC architecture as opposed to a vector one.

In another case study we isolated errors that occurred when a photo-chemical pollution model was ported from one sequential workstation to another [1]. In this case the code was identical but the two machines produced different answers. The errors were finally attributed to the different behaviour of a key library function, which returned slightly divergent results on the two platforms.

In a more recent case study we isolated errors that occurred when a program was rewritten from C into another language, ZPL, for execution on a parallel platform [24]. This case study was interesting because even though the two codes were producing slightly different answers, the divergence was attributed to different floating point precision. However by using Guard it was possible to show that there were actually four independent coding errors—three in the new ZPL program, and surprisingly, one in the original C code.

All of these case studies have highlighted the power of relative debugging in the process of developing scientific codes. We believe that many of the same issues will arise when migrating scientific software to the new .NET Framework and that Guard will be able to play an important role in assisting this process.

4. The .NET Framework

The Microsoft .NET Framework represents a significant change to the underlying platform on which Windows applications run [17]. The .NET Framework defines a runtime environment that is common across all languages. This means that it is possible to write applications in a range of languages, from experimental research ones to standard production ones, with the

expectation that similar levels of performance and efficiency will be achieved. An individual program can also be composed of modules that are written in different languages, but that interoperate seamlessly. All compilers that target the .NET environment generate code in an Intermediate Language (IL) that conforms to a CLS. The IL is in turn compiled into native code using a just-in-time compilation strategy. These features mean that the .NET Framework should provide an efficient platform for developing computational models.

The Web Services features of .NET also offer significant scope for scientific applications. At present most computational models are built as single monolithic codes that call library modules using local procedure calls. More recent developments such as the NetSolve and NEOS application servers have provided an exception to this strategy. These services provide complex functions such as matrix algebra and optimisation algorithms using calls to external servers. When an application uses NetSolve, it calls a local ‘stub’ module that communicates with the NetSolve server to perform some computation. Parameters are sent via messages to the server and results are returned the same way. The advantage of this approach is that application programmers can benefit by using ‘state of the art’ algorithms on external high-performance computers without the need to run the codes locally. Further, the load balancing features of the systems are able to allocate the work to servers that are most lightly loaded. The major drawback of external services like this is that the application must be able to access to required server and so network connectivity becomes a central point of failure. Also, building new server libraries is not easy and requires the construction of complex web hosted applications. The .NET Framework has simplified the task of building such servers using its Web Services technology. Application of Web Services to science and engineering programs is an area of interest that requires further examination.

Visual Studio.NET (VS.NET) is the preferred code development environment for the .NET Framework. The VS.NET environment represents a substantial change to previous versions of Visual Studio. Older versions of Visual Studio behaved differently depending on the language being supported—Visual Basic used a different set of technologies for building applications than Visual C++. The new VS.NET platform

has been substantially re-engineered and as a consequence languages are now supported in a much more consistent manner.

VS.NET also differs from previous versions by exposing many key functions via a set of remote APIs known as ‘automation’. This means that it is possible to write a third party package that interacts with VS.NET. For example, an external application can set breakpoints in a program and start the execution without user interaction. A separate Software Development Kit (SDK) called VSIP (Visual Studio Integration Program) makes it possible to embed new functions directly into the environment. This allows a programmer to augment VS.NET with new functionality that is consistent with other functions that are already available and operates seamlessly with the user interface. This feature has allowed us to integrate a version of Guard with Visual Studio as discussed in the next section.

5. Architecture of guard

Fig. 1 shows a simplified schematic view of the architecture of Guard under VS.NET. VS.NET is built around a core ‘shell’ with functionality being provided by commands that are implemented by a set of ‘packages’. These packages are conventional COM objects that are activated as a result of user interaction (such as menu selection) within VS.NET, and also when various asynchronous events occur. This component architecture makes it possible to integrate new functionality into the environment by loading additional packages.

Debugging within the VS.NET environment is supported by three main components. The Debugger package provides the traditional user interface commands such as ‘Go’, ‘Step’, ‘Set Breakpoint’, etc. that appear in the user interface. This module communicates with the Session Debug Manager, which in turn provides a multiplexed interface into one or more per-process Debug Engines. The Debug Engines implement low-level debug functions such as starting and stopping a process, setting breakpoints, and providing access to the state of the process. Debug Engines can cause events to occur in response to conditions such as a breakpoint being reached and these are passed back through the Session Debug Manager to registered event handlers.

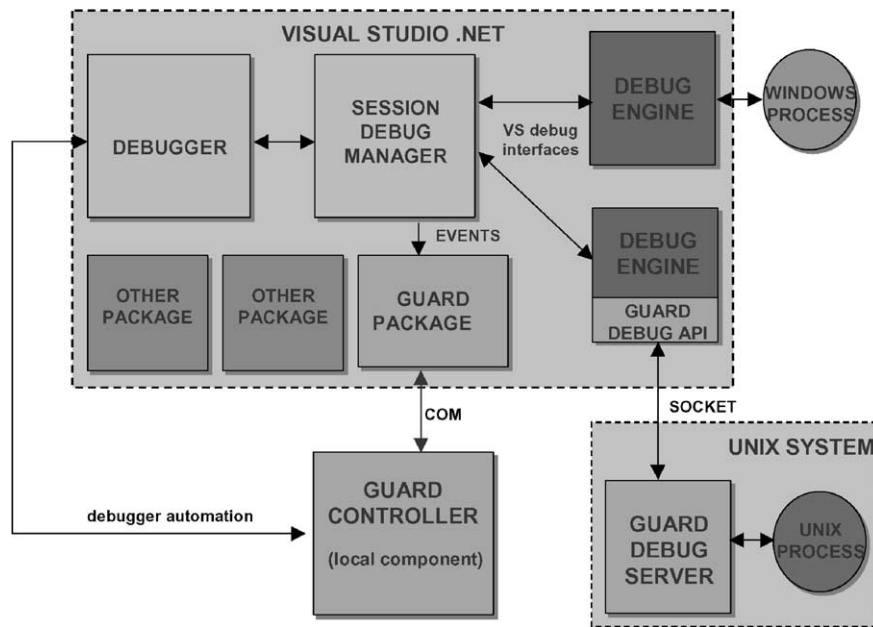


Fig. 1. Guard architecture.

Each Debug Engine is responsible for controlling the execution of a single process. The VS.NET architecture also supports the concept of remote debugging, so a process being debugged may be running on a remote Windows system.

The VS.NET implementation of Guard consists of three main components. A package is loaded into the VS.NET shell that incorporates logic to respond to specific menu selections and handle debugger events. This package executes in the main thread of the shell and therefore has had to be designed to avoid blocking for any extended time period. The main relative debugging logic is built into a local COM component called the Guard Controller. This is a separate process that provides a user interface for managing assertions and a dataflow interpreter that is necessary to implement relative debugging. Because the Guard Controller runs as a separate process it does not affect the response of the main VS.NET thread. The Guard Controller controls the programs being debugged using the VS.NET automation interface. We have also built a Debug Engine that is able to control a process running on an external UNIX platform. This works by communicating with the remote debug server developed for the original UNIX version of Guard using a TCP/IP socket and

a custom protocol. The UNIX debug server, based on the GNU GDB debugger, is available for most variants of UNIX, and provides basic debug functions, including process startup to Guard. We have modified GDB to provide support for an Architecture Independent Format (AIF) [24] for data structures, which means it is possible to move data between machines without being concerned about different architectural characteristics, such as word size, endian'ness, etc. AIF also facilitates machine independent comparison of data structures. It is the addition of this Debug Engine that allows us to compare programs executing on Windows and UNIX platforms.

The architecture of Guard is consequently very flexible and allows debugging distributed processes as well as monolithic codes. For example, if an application were decomposed into a number of distinct processes, possibly implemented as Web services, then it would be possible to specify assertions between these individual components and the original sequential code version.

Fig. 2 shows a screen dump of Guard running under VS.NET. When a user wishes to compare two running programs they must first be loaded into a VS.NET 'solution' as separate 'projects'. The solution is then

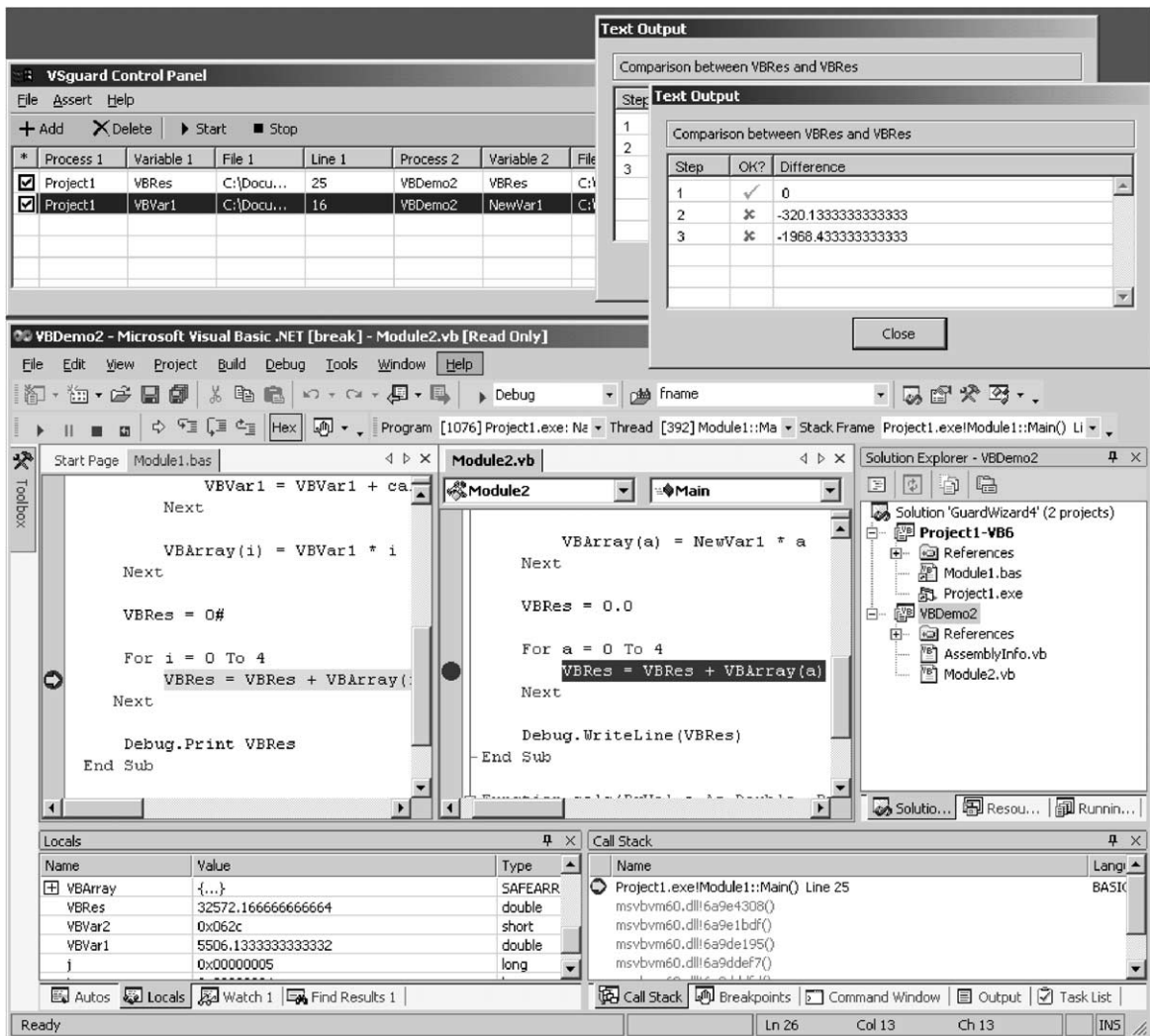


Fig. 2. Guard controller and VS.NET.

configured to start both programs running at the same time under the control of individual Debug Engines. The source windows of each project can then be tiled to allow both to be displayed at once.

A user creates an assertion between the two programs using the Guard Controller, which is started by selecting the 'VSGuard' item from the 'Tools' menu. The Guard Controller has a separate Control Panel window as shown. An assertion is created in a few simple steps. A new, empty, assertion is created by selecting the 'Add' button. Guard displays the dialog

box shown in Fig. 3, which allows the user to enter the information necessary to create an assertion. The left-hand side of the assertion can be automatically populated with the variable name, line number, source file and program information by selecting the required variable in the appropriate source window and then using a single right-mouse click. The right-hand side of the assertion can be filled in using the same technique in the other source window. Finally the user is able to specify properties about the assertion such as the error value at which output is generated, when the

Fig. 3. New assertion dialog.

debugger should be stopped and the type of output to display. The user can create any number of assertions by repeating this process and then launch the programs using the 'Start' button on the Control Panel.

Before commencing execution Guard automatically sets breakpoints at the locations in the source files specified by the assertions. During execution Guard will extract the contents of a variable when its corresponding breakpoint is reached and then perform a comparison once data from each half of the assertion has been obtained. Once the appropriate error threshold has been reached (as specified in the assertion), Guard will either display the results in a separate window or stop the debugger to allow interactive examination of the programs' state. Guard currently supports a number of display types including text, bitmaps and the ability to export data into a visualisation package.

6. Implementation issues

While the VS.NET debugger architecture has been designed with the ability to manage and debug multiple processes, it is not a true multi-process debugger. This is because:

- (a) the debugger does not provide control operations, such as start/restart and single step on a

per-process basis. Instead these functions only operate on all processes collectively; and

- (b) the debugger is constrained to operate in one of two modes where a break event (such as a breakpoint being reached) in one process stops all processes together, or a break event only stops execution of the current process but the debugger is unable to access the other process to obtain state information.

In contrast, the existing Guard architecture assumes that independent control of individual processes is provided by the debugger infrastructure. To address this issue we have modified Guard so that processes are restarted as soon as possible after a breakpoint is reached and data has been extracted. However, because a restart command is issued to all processes Guard must also keep a record of the state of each process so that it can ensure that restart commands are only issued at the appropriate time. The result of this modification is some loss of functionality over the existing UNIX version. In particular it is not possible to stop both programs at a known location when an assertion threshold is exceeded, complicating the manual debugging that might occur after an assertion has triggered.

Our original intention was to integrate both the user interface and a dataflow interpreter into a single multi-threaded package in VS.NET. However

due to limitations in relation to the thread safety and re-entrancy of the VS.NET shell we were not successful in this approach. Instead, by separating these parts of the debugger into a local component the threading issues were effectively eliminated.

Another implementation issue arose because of our use of the debugger automation interface that is provided by Visual Studio. This interface is used to extract the contents of program variables when a process is stopped at a breakpoint. Unlike our UNIX debug server API, which is designed to transfer large data structures efficiently, the automation interface is only able to extract data from a simple object with each call. This means that extracting data from complex objects such as arrays can be very time consuming since repeated calls must be made for each element. To solve this problem our package must expose its own interface to the low-level data access facilities provided by the Debug Engines.

A further consideration was in relation to the need to use AIF in the Guard debugger. If Guard operated exclusively in the WIN32 environment then it would be feasible to perform all comparisons using only the native data format and avoid the overhead incurred by using the AIF library routines. However since we wish to use Guard to compare data between UNIX, WIN32 and .NET systems we must employ an architecture neutral format. Because VS.NET employs its own internal format, data arriving at the Debug Engine from a UNIX debug server must first be converted into this format. Once Guard receives the data via the debugger automation interface it must then be converted back into AIF before being processed by the dataflow interpreter. This results in extra overheads because of the dual conversion but simplifies the implementation since we do not need to modify the dataflow interpreter code. We plan to investigate architectural modifications to Guard that remove the need for the multiple format conversions.

One final issue is in relation to the asynchronous behaviour of programs being debugged under the VS.NET environment. We have occasionally observed situations where one process of a multi-process debug session receives significantly more execution time than the others, particularly in cross-language situations or where VS.NET is controlling both managed (.NET) and un-managed (legacy) code. Since the dataflow architecture employed by Guard is designed

to deal with this situation, it is not a serious issue, although it can only do so for a finite time before all its internal buffers become filled. We will be monitoring this situation to see if the problem manifests in later versions of .NET and if strategies need to be incorporated into Guard to deal with the issue.

In spite of these difficulties, the implementation has been fairly smooth and a prototype version of Guard has been produced. Visual Studio is one of the few interactive environments that have been designed with the goal of incorporating third party packages [11,13], and we have demonstrated that this integration is possible. Specifically, we have been successful in incorporating a tool that will support the migration of applications to the new .NET Framework.

7. The ‘Earth’ case study

As discussed in [Section 1](#), porting a code from one platform to another poses significant challenges for the programmer. Many of these challenges are present even when the application is only migrated from one version of a language to another, regardless of whether the platform also changes. In this section we illustrate the power of relative debugging using a small scientific program called ‘Earth’, written in Visual Basic. The problems we experienced in doing this were no different from problems we have experienced in the past when we moved code from one platform and operating system to another.

‘Earth’ is a free program that uses the VSOP87 planetary theory to compute the heliocentric ecliptic longitude (L), latitude (B) and the distance to the sun (R) of the planet Earth over a period of several thousands of years [22]. It is based on the same mathematical formulations used to compute long-term, high-precision, heliocentric orbital positions of the planets when preparing astronomical almanacs. Originally written in Visual Basic 5 and Visual Basic 6, we decided to upgrade it to Visual Basic.NET. Visual Basic 6 and Visual Basic.NET have minor, but subtle, syntactic and semantic differences. Whilst the example is performed on the same operating system, it highlights many of the same issues that arise when a program is moved across platforms, and serves as an adequate illustration of how Guard can be applied. The program was converted using a special wizard

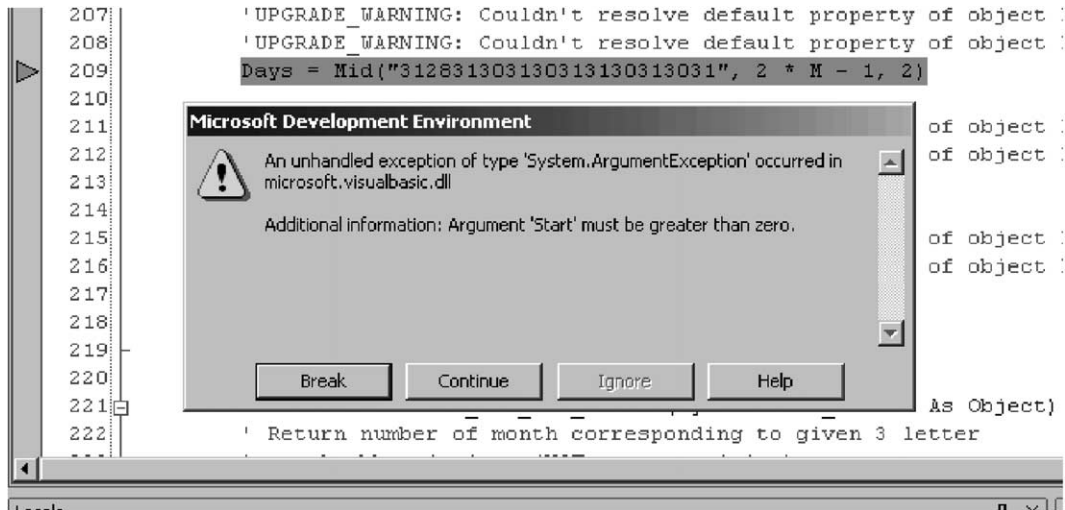


Fig. 4. Exception generated in Earth.

provided by Microsoft, and whilst there were a number of issues that required inspection, the wizard seemed to make sensible decisions during the translation process. In spite of this, the code did not run correctly and consequently required debugging.

7.1. Problem 1

Whilst Earth compiled without errors, it generated a runtime error during execution, as shown in Fig. 4.

Step 1. Using Guard to place an assertion on the statement in error showed that one of the arguments to the Mid function was different between the two versions, in particular, the variable M had a value of 1 in the Visual

Basic 6 code and 0 in the Visual Basic.NET version. Fig. 5 shows the code used in the derivation of M.

- Step 2. Placing assertions on Q and Month_Year_BCAD indicated that they were both wrong. Since Month_Year_BCAD was passed as an argument, we next inspected the location of the call to DAYS_IN_MONTH_OF. Fig. 6 shows the code fragment responsible.
- Step 3. Placing assertions on M, Y, MonthSelect.Text and Year_Renamed.Text indicated that they were all incorrect, more over MonthSelect.Text and Year_Renamed.Text were actually initialised.
- Step 4. Closer inspection of the code showed that there were multiple calls to the function

```
Public Function DAYS_IN_MONTH_OF(ByRef Month_Year_BCAD
    As Object) _ As Object
    Dim Leap, Q, Days, Y As Object
    Dim M As Integer
    Q = Trim(Month_Year_BCAD)
    M = MONTH_NUM_FOR_ABBREV(Left(Q, 3))
    Y = Trim(Mid(Q, 4, 16))
    Days = Mid("312831303130313130313031", 2 * M - 1, 2)
    If IS_LEAP_YEAR(Y) = True And M = 2 Then Days = 29
    DAYS_IN_MONTH_OF = Days
End Function
```

Fig. 5. Code fragment for computing M.

```

Private Sub ADJUST_MONTH_LENGTH ()
    :
    :
    :
    D = Val(Trim(DaySelect.Text))
    M = Trim(MonthSelect.Text)
    Y = Trim(Year_Renamed.Text)
    If BCOption.Checked = True Then
        Y = Y & " BC"
    Else
        Y = Y & " AD"
    End If
    NDays = Val(DAYS_IN_MONTH_OF(M & " " & Y))

```

Fig. 6. Code fragment for calling DAYS_IN_MONTH_OF.

ADJUST_MONTH_LENGTH, some of which were as a result of handling changed field events. Using the Visual Studio Call Stack view, we observed that ADJUST_MONTH_LENGTH had in fact been called from different places in the two versions of the program.

A code search showed that MonthSelect.Text and Year_Renamed.Text were initialised in the form-load event handler, but that the .NET version of the program was invoking a changed event handler as a side effect of initialising the form. This in turn caused the ADJUST_MONTH_LENGTH code to be executed earlier than in the Visual Basic 6 version, and thus before it was properly initialised.

7.1.1. The root cause of this error was different event ordering in Visual Basic 6 and Visual Basic.NET

Even a simple error such as this illustrates the power of relative debugging. Using conventional techniques we would have quickly been able to determine that the value of M was incorrect, since it was generating an invalid value for the Mid function. However at this point we would have had no idea whether the root of the problem arose in the MONTH_NUM_FOR_ABBREV function or in the value of Q. In tracking the cause we might have spent considerable time examining code for errors, even though it was actually correct. Using relative debugging we immediately know where the source of the error is located.

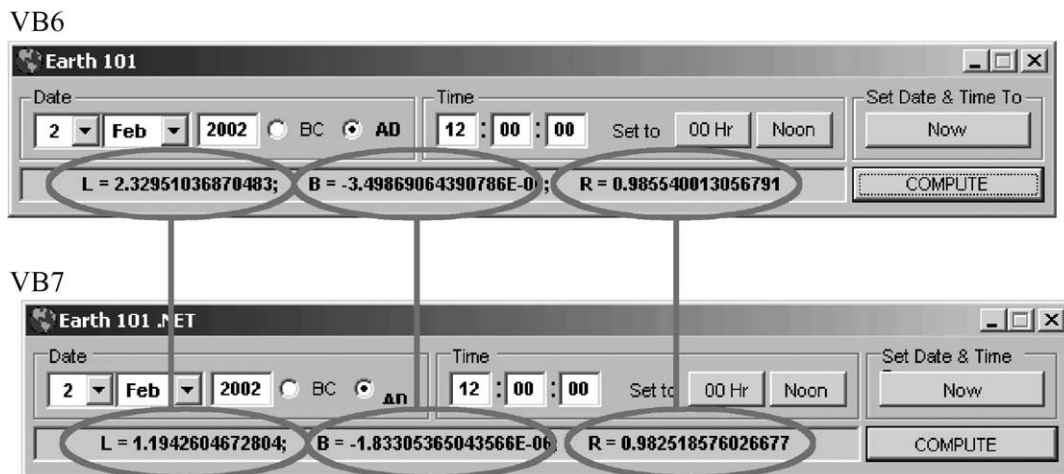


Fig. 7. Values of L, B and R in error.

```

Private Sub ComputeButton_Click(ByVal eventSender As System.Object,
ByVal eventArgs As System.EventArgs) Handles
    ComputeButton.Click
    Dim Q As Object
    Q = JDE_FOR(INTERFACE_DATE, INTERFACE_TIME)
    Q = EARTH_LBR_FOR(Q)
    Text2.Text = Q
End Sub

```

Fig. 8. Code fragment for computing L , B and R .

```

Public Function JDE_FOR(ByRef Date_String As Object,
    ByRef Time_String As Object) As Object
    Dim Q, W As Object
    W = JD_NUM_FOR(Trim(Date_String))
    Q = Trim(Time_String)
    Q = (Val(Left(Q, 2)) * 3600# + Val(Mid(Q, 4, 2)) * 60# +
        Val(Mid(Q, 7, 16))) / 86400#
    JDE_FOR = W + Q
End Function

```

Fig. 9. Code fragment for JDE_FOR.

```

Public Function JD_NUM_FOR(ByRef Dd_Mmm_Yyyy_BCAD As Object)
    As Object
    ...
    Date_String = Trim(UCase(Dd_Mmm_Yyyy_BCAD))
    Q = ""
    For Pointer = 1 To Len(Date_String)
        Q1 = Mid(Date_String, Pointer, 1)
        If Q1 <> " " Then Q = Q & Q1
    Next Pointer
    Date_String = Q
    DD = Val(Q)
    Pointer = InStr(1, Q, DD) + Len(DD)
    ...
    MMM = Mid(Q, Pointer, 3): Pointer = Pointer + 3
    MM = Int(1 + ((InStr(1, "JANFEBMARAPRMAYJUNJULAUGSEP OCTNOVDEC",
        MMM) - 1) / 3))
    ...
    JD = DD + Int(367 * (MM + (Q * 12) - 2) / 12)
        + Int(1461 * (YYYY + 4800 - Q) / 4) - 32113
    ...
    JD_NUM_FOR = JD - 0.5
End Function

```

Fig. 10. Code fragments for JD_NUM_FOR.

```
Pointer = InStr(1, Q, DD) + Len(CType(DD, System.String))
```

Fig. 11. Fixing the coding error.

7.2. Problem 2

On correcting the first problem, the program ran without error. However the values being computed were wrong, as shown in Fig. 7.

- Step 1. Using Guard to place assertions on Q, INTERFACE_DATE and INTERFACE_TIME (see Fig. 8) we determined that the error was contained in the JDE_FOR function (shown in Fig. 9).
- Step 2. Placing assertions on Date_String, W and Q indicated that the routine JD_NUM_FOR was in error. The relevant code is shown in Fig. 10.
- Steps 3–5. Placing assertions on the result of JD_NUM_FOR and the define points of variables MM, MMM, Pointer, Q and DD allowed us to trace back through the expressions and determine that DD was correct but Len(DD) was incorrect. On closer inspection it transpired that in the Visual Basic 6 version of the code, the Len function only takes a string argument, and thus when it is passed a variant of type double this is first converted to a character string, and then the length of that string is returned. However, in Visual Basic.NET, the Len function takes an object as a parameter and then it returns the length of the object—in this case a value of 8. The code can be corrected by explicitly converting DD to a string as shown in Fig. 11.

7.2.1. The root cause of this error was different handling of variant parameters with built-in functions

An important point with both of these errors, is that whilst a conventional debugger would have allowed us to explore whether values were correct or not, it would also have required us to have a good understanding of the values that were expected and the algorithms being employed. On the other hand,

with relative debugging we did not need to have such a mental model, and could focus instead on just comparing one program with another and tracing the define points of variables. When the data structures are complex and large, this becomes a very significant advantage over conventional approaches. Moreover, the errors were located very quickly and efficiently.

8. Future work and conclusions

It is far too early to claim that .NET is a suitable platform for scientific computation since it has only been released for a short time and there are few commercial codes available, and virtually no scientific ones. As discussed in Section 1 we believe that .NET offers a number of potential benefits for large numeric models. However, the execution environment is very different from other platforms and so it is critical that as many tools as possible are available to facilitate the transition of existing legacy software. Guard is one such tool because it allows a user to compare two executing programs simultaneously on different platforms.

Whilst the implementation of Guard under UNIX alone is mature and has been used on many case studies, the current version under VS.NET is still in a pre-beta testing phase. Specifically, the control of programs by Visual Studio under UNIX at this stage is rudimentary and this is why the case study presented here focussed entirely on the Windows operating system. We are also planning a number of extensions that will be required if Guard is to be of practical use in supporting the migration to .NET. The current user interface is fairly simple and must be made more powerful if it is to be applied to large programs. At present only simple data types and arrays are supported. We need to extend this to encompass the range of types found in scientific codes, such as structures and other complex types. Assertions need to be able to be saved and restored when the environment is restarted, and assertions should employ symbolic markers which are independent of the actual numeric line numbers.

We are also planning to integrate Guard into Source Safe [10], Microsoft's equivalent of SCCS [18] or RCS [23] making it possible to compare one version of a program with previous versions automatically. We have already experimented with a version of Guard under UNIX that provides explicit support for parallel programming [24], and we plan to enhance the support for multi-process programs in the Visual Studio version to make it feasible to debug programs in running on a cluster of Windows machines. Finally, we are working on a new version of Guard that attempts to perform much of the generation and refinement of assertions automatically. Whilst this project is in the early stages, it appears that using powerful data flow analysis of the two programs would allow us to generate and refine a number of the assertions without user involvement.

Acknowledgements

This work has been funded by grants from the Australian Research Council and Microsoft Corporation. We wish to acknowledge the support of a number of individuals at Microsoft for their assistance on various issues related to Visual Studio and .NET. Particular thanks go to Todd Needham, Dan Fay and Frank Gocinski. We also wish to acknowledge our colleagues, Professor Christine Mingins, Professor Bertrand Meyer and Dr. Damien Watkins for many helpful discussions. Some of the coding for the UNIX debug engine was performed by Le Phu Dung, and the 'Earth' case study was performed by Tim Ka-chung Ho and Clement Chu.

References

- [1] D. Abramson, I. Foster, J. Michalakes, R. Susic, Relative debugging: a new paradigm for debugging scientific applications, *Commun. Assoc. Comput. Mach.* 39 (11) (1996) 67–77.
- [2] D. Abramson, I. Foster, J. Michalakes, R. Susic, Relative debugging and its application to the development of large numerical models, in: *Proceedings of the IEEE Supercomputing*, San Diego, CA, December 1995.
- [3] D. Abramson, R. Susic, A debugging and testing tool for supporting software evolution, *J. Autom. Softw. Eng.* 3 (1996) 369–390.
- [4] D. Abramson, R. Susic, A debugging tool for software evolution, in: *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE-95)*, Toronto, Ont., Canada, July 1995, pp. 206–214;
- [5] D. Abramson, R. Susic, A debugging tool for software evolution, in: *Proceedings of the Second Working Conference on Reverse Engineering*, Toronto, Ont., Canada, July 1995.
- [6] D. Abramson, R. Susic, G. Watson, Implementation techniques for a parallel relative debugger, in: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, October 20–23, 1996, Boston, MA.
- [7] T. Bemmerl, R. Wismüller, On-line distributed debugging on scaleable multicomputer architectures, *High Performance Computing and Networking*, vol. II: Networking Tools, Lecture Notes in Computer Science, vol. 797, Springer, Berlin, April 1994, pp. 394–400.
- [8] H. Casanova, J. Dongarra, NetSolve: a network server for solving computational science problems, *Int. J. Supercomput. Appl. High Perform. Comput.* 11 (3) (1997) 212–223.
- [9] D. Cheng, R. Hood, A portable debugger for parallel and distributed programs, in: *Proceedings of the Supercomputing'94*, November 1994, pp. 723–732.
- [10] J. Czyzyk, J. Owen, S. Wright, Optimization on the internet, *OR/MS Today*, October 1997.
- [11] <http://www.seg.org/research/3Dmodel/SEPformat.html><http://msdn.microsoft.com/ssafe/>.
- [12] <http://www.eclipse.org>.
- [13] <http://www.lahey.com/netwtp1.htm>.
- [14] <http://www.sun.com/forte/>.
- [15] L. Snyder, *A Programmer's Guide to ZPL*, MIT Press, Cambridge, MA, 1999.
- [16] T.J. LeBlanc, J.M. Mellor-Crummey, Debugging parallel programs with instant replay, *IEEE Trans. Computers* C-36 4 (1987) 471–482.
- [17] J. May, F. Berman, Panorama: a portable extensible parallel debugger, in: *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA, May 1993, pp. 96–106.
- [18] B. Meyer, .NET is coming, *IEEE Comput.* 34 (8) (2001) 92–97.
- [19] *Programming Utilities and Libraries*, Sun Release 4.1, Sun Microsystems, 1988.
- [20] N. Ramsey, D. Hanson, A retargetable debugger, in: *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, ACM, 1992, pp. 22–31.
- [21] R. Susic, D. Abramson, Guard: a relative debugger, *Softw. Pract. Exp.* 27 (2) (1997) 185–206.
- [22] R. Stallman, Debugging with GDB—The GNU Source Level Debugger, 4.12 ed., Free Software Foundation, January 1994.
- [23] J. Tanner, A. Mason, <http://FreeVBCode.com>.
- [24] W. Tichy, RCS—a system for version control, *Softw. Pract. Exp.* 15 (7) (1985) 637–654.
- [25] G. Watson, D. Abramson, Relative debugging for data parallel programs: a ZPL case study, *IEEE Concurrency* 8 (4) (2000) 42–52.
- [26] R. Wismüller, M. Oberhuber, J. Krammer, Interactive debugging and performance analysis of massively parallel applications, *Parallel Comput.* 22 (3) (1996) 415–442.

David Abramson has been involved in computer architecture and high-performance computing research since 1979. Previous to joining Monash University in 1997, he has held appointments at Griffith University, CSIRO, and RMIT. At CSIRO he was the program leader of the Division of Information Technology High-Performance Computing Program, and was also an adjunct Associate Professor at RMIT in Melbourne. He was also a program manager in the Co-operative Research Centre for Intelligent Decisions Systems. Abramson is currently the head of the School of Computer Science and Software Engineering (CSSE) at Monash University, Australia. CSSE consists of over 80 academic staff members across two campuses. He is a project leader in the Co-operative Research Centre for Distributed Systems Nimrod Project and also Chief Investigator on an ARC funded research project called Guard, a relative debugger. Abramson has chaired a number of international conferences, including the prestigious ACM International Symposium on Computer Architecture in 1992. He has published over 100 papers and technical documents. He has given seminars and received awards around

Australia and internationally and has received nearly \$2 million in research grants. He is a co-founder of Active Tools P/L with Dr. Rok Sasic, a company which was established to commercialise the Nimrod project and Guardsoft, a company focused on commercialising the Guard project. Abramson's current interests are in high-performance computer systems design and software engineering tools for programming parallel and distributed super-computers.

Greg Watson is a technical staff member in the Advanced Computing Lab at Los Alamos National Laboratory. His research interests focus on tools for parallel computers, program debugging, distributed computing and operating systems. He received his BS in computer science from the University of Tasmania and his PhD from Monash University. Prior to his current position, Greg worked as a senior research fellow in the School of Computer Science and Software Engineering at Monash University. He is a member of the IEEE, Immediate Past President of the Internet Society of Australia and Co-Chair of the Australian Domain Administration.