Keywords
Abstract

Relative Debugging is a paradigm that assists users to locate errors in programs that have been corrected or enhanced. In particular, the contents of key data structures in the development version are compared with the contents of the corresponding data structures, in an existing version, as the two programs execute. If the values of two corresponding data structures differ at points where they should not, an error may exist and the user is notified.

Relative Debugging requires users to identify the corresponding data structures within the two programs, and the locations at which the comparisons should be performed. To quickly and effectively identify useful data structures and comparison points requires that users have a detailed knowledge of the two programs under consideration. Without a detailed knowledge of the two programs, the task of locating useful data structures and comparison points can quickly become a difficult and time consuming process. Prior to the research detailed in this thesis, the Relative Debugging paradigm did not provide any assistance that allowed users to quickly and effectively identify suitable data structures and program points that will help discover the source of an error.

Our research efforts have been directed at enhancing the Relative Debugging paradigm. The outcome of this research is the discovery of techniques that empower Relative Debugging users to become more productive and allow the Relative Debugging paradigm to be significantly enhanced. Specifically, the research has resulted in the following three contributions:

1. A Systematic Approach to Relative Debugging,
2. Data Flow Browsing for Relative Debugging,
3. Automatic Relative Debugging.

These contributions have enhanced the Relative Debugging paradigm and allow errors to be localized with little human interaction. Minimizing the user’s involvement reduces the cost of debugging programs that have been corrected or enhanced, and has a significant impact on current debugging practices.
# Table of Contents

1. **Introduction** ........................................................................................................................................... 19

2. **Background and Motivation** .................................................................................................................... 27
   2.1  Software Maintenance and Evolution ....................................................................................... 28
       2.1.1  The Cost of Evolution ........................................................................................................ 29
       2.1.2  The Evolutionary Process and Supporting Tools ......................................................... 32
       2.1.3  Current Debugging Support for Software Evolution ...................................................... 33
           2.1.3.1  Guard: A Relative Debugger ................................................................................ 34
           2.1.3.2  p2d2: A Portable Distributed Debugger .......................................................... 37
       2.2  Automatic Debugging .................................................................................................................. 38
           2.2.1  Shapiro’s Interactive Diagnosis Algorithm ............................................................... 38
           2.2.2  vpoiso ............................................................................................................................ 39
           2.2.3  DynaDiff ....................................................................................................................... 40
           2.2.4  Delta Debugging ......................................................................................................... 40
           2.2.5  WHITHER .................................................................................................................... 43
       2.3  Automatic Relative Debugging .................................................................................................. 43

3. **A Systematic Approach to Relative Debugging** ......................................................................................... 45
   3.1  The Relative Debugging Paradigm ............................................................................................... 46
       3.1.1  Imperative Assertions ....................................................................................................... 49
       3.1.2  Declarative Assertions ....................................................................................................... 50
       3.1.3  A Systematic Approach to Relative Debugging ............................................................. 51
           3.1.3.1  Example .................................................................................................................. 52
       3.2  Enhancing the Relative Debugging Paradigm ............................................................................ 53

4. **Data Flow Browsing for Relative Debugging** .......................................................................................... 55
   4.1  What is a Data Flow Browser ? .................................................................................................... 55
   4.2  Why Another Data Flow Browser ? .......................................................................................... 56
       4.2.1  Example .......................................................................................................................... 57
   4.3  What’s Different about a Relative Debugging Data Flow Browser ? ....................................... 58
       4.3.1  Efficiency ....................................................................................................................... 58
       4.3.2  Multilingual ................................................................................................................... 59
       4.3.3  Usability ........................................................................................................................ 60
   4.4  Design of a Relative Debugging Data Flow Browser ............................................................... 60
       4.4.1  Efficiency ....................................................................................................................... 60
           4.4.1.1  Use Define Chains ......................................................................................... 61
4.4.1.2 Demand Driven Analysis ...................................................... 62
4.4.2 Multilingual ................................................................. 64
  4.4.2.1 Related Work ............................................................... 64
4.4.3 Usability ................................................................. 64
4.4.4 Algorithm ................................................................. 65
  4.4.4.1 Example ................................................................. 67
4.5 DUCT: A Data Flow Browser for Relative Debugging .......... 69
  4.5.1 Architecture ................................................................. 69
  4.5.2 User Interface ................................................................. 71
    4.5.2.1 DUCT ................................................................. 71
    4.5.2.2 VSGuard ................................................................. 73
  4.5.3 Implementation Details .................................................. 74
    4.5.3.1 Generating Intraprocedural UD Chains .................. 75
    4.5.3.2 Generating Interprocedural UD Chains ............... 86
4.6 Case Study .................................................. 89
4.7 Conclusion .................................................. 94

5 Automatic Relative Debugging ................................................. 95
  5.1 Automatically Generating Assertions ...................... 97
    5.1.1 Generating Intraprocedural Assertions with Static Analysis .... 101
      5.1.1.1 Assertion Graph ........................................... 102
      5.1.1.2 Multiple Definitions ...................................... 104
      5.1.1.3 Control Variables .......................................... 105
      5.1.1.4 False Positives ............................................. 106
      5.1.1.5 Suspect Assertions ........................................ 109
    5.1.2 Generating Interprocedural Assertions with Dynamic Analysis .... 110
      5.1.2.1 Function Assertions ....................................... 112
      5.1.2.2 Incoming Parameter Assertions ...................... 112
      5.1.2.3 Selecting Continuing Definitions .................... 113
      5.1.2.4 Call Sensitive Assertions ............................. 116
      5.1.2.5 Iterative Code ............................................. 117
  5.2 Unable to Generate Assertions when the Programs Differ .... 119
    5.2.1 Cloning for Automatic Relative Debugging ............... 120
    5.2.2 Renamed Variables ............................................. 123
    5.2.3 Different Data Flow ............................................ 124
  5.3 Automatic Relative Debugging Summary .................... 127
  5.4 Autoguard: An Automatic Relative Debugger ............. 127
    5.4.1 Architecture .................................................. 127
# Table of Figures

| Figure 2-1: Cost of software maintenance as reported by several surveys. | 31 |
| Figure 3-1: The Relative Debugging Paradigm. | 48 |
| Figure 3-2: Creating assertions using the Relative Debugging Methodology | 53 |
| Figure 4-1: Using data flow information to debug a program | 57 |
| Figure 4-2: Slicing tools | 59 |
| Figure 4-3: Example UD Chain | 61 |
| Figure 4-4: Static program slice with respect to variable x on the last line | 63 |
| Figure 4-5: Dynamic program slice with respect to variable x on the last line and pred = true. | 63 |
| Figure 4-6: Data structures required to construct UD chains | 66 |
| Figure 4-7: Program for UD Chain construction example | 68 |
| Figure 4-8: Control flow graph for program listed in Figure 4-7. | 68 |
| Figure 4-9: Extending Microsoft Visual Studio .NET with DUCT and VSGuard. | 70 |
| Figure 4-10: Example DUCT screenshot | 72 |
| Figure 4-11: VSGuard Edit Assert Dialog Box. | 73 |
| Figure 4-12: VSGuard Assertion Results Window | 74 |
| Figure 4-13: Microsoft Intermediate Language (MSIL) example | 75 |
| Figure 4-14: MSIL control transfer instructions | 78 |
| Figure 4-15: MSIL store instructions | 78 |
| Figure 4-16: Indirect store instruction requiring symbolic execution of the abstract stack machine. | 79 |
| Figure 4-17: Example C# method | 81 |
| Figure 4-18: Extracted method details for method listed in Figure 4-17 | 82 |
| Figure 4-19: Control flow graph and extracted method details for program listed in Figure 4-17. | 83 |
| Figure 4-20: Definitions located by DUCT in the program listed in Figure 4-17 (screenshot). | 84 |
| Figure 4-21: Definitions located by DUCT in a VB .NET program (screenshot). | 85 |
| Figure 4-22: MSIL call instructions. | 86 |
| Figure 4-23: DUCT case study - result produced by the original Earth VB program. | 89 |
| Figure 4-24: DUCT case study - result produced by the ported Earth VB.NET program. | 89 |
| Figure 4-25: DUCT case study - code producing the erroneous result. | 90 |
| Figure 4-26: DUCT case study – using DUCT to locate definitions for the input parameter Q | 90 |
| Figure 4-27: DUCT case study – code containing definitions located by DUCT | 91 |
| Figure 4-28: DUCT case study - VSGuard result showing that variable W differed. | 91 |
| Figure 4-29: DUCT case study - using DUCT to locate definitions for the input parameter W | 92 |
| Figure 4-30: DUCT case study - JD_NUM_FOR function. | 93 |
| Figure 4-31: DUCT case study - required correction. | 93 |
| Figure 5-1: Automatic Relative Debugging approach. | 100 |
| Figure 5-2: Automatically generated assertions. | 101 |
| Figure 5-3: Assertion graph for the assertions generated in Figure 5-2. | 103 |
| Figure 5-4: Multiple definitions - program 1 | 104 |
| Figure 5-5: Multiple definitions - program 2 | 104 |
Figure 6-21: Autoguard case study 2 - errors in the ported Fortran program output. ......................... 162
Figure 6-22: Autoguard case study 2 - Autoguard Control Panel showing errors during iteration 3. 163
Figure 6-23: Autoguard case study 2 - Autoguard Control Panel showing errors during iteration 4. 164
Figure 6-24: Autoguard case study 2 – assertion graph for iteration 4. ........................................... 165
Figure 6-25: Autoguard case study 2 – values captured by assertion GA0017. ................................. 165
Figure 6-26: Autoguard case study 3 - original PREDICT. ................................................................. 169
Figure 6-27: Autoguard case study 3 – correct position of the OSCAR-11 satellite. ............................ 170
Figure 6-28: Autoguard case study 3 - ported PREDICT. ................................................................. 170
Figure 6-29: Autoguard case study 3 – incorrect position of the OSCAR-11 satellite. ......................... 171
Figure 6-30: Autoguard case study 3 – Autoguard results for iterations 1 through to 5. ..................... 172
Figure 6-31: Autoguard case study 3 - values captured by call sensitive assertion GA0004. ............ 173
Figure 6-32: Autoguard case study 3 - values captured by non call sensitive assertion GA0004. ..... 174
Figure 6-33: Autoguard case study 3 – assertion graph for iteration 4. ............................................ 175
Figure 6-34: Autoguard case study 3 – Autoguard results for iterations 6 and 7. .............................. 176
Figure 6-35: Autoguard case study 3 - Iteration summary for iterations 1, 2 and 3. .......................... 178
Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Aaron Searle
March 2007
Acknowledgements

First and foremost, I would like to thank my mother for her unconditional support. Without her support, encouragement, and prays, my PhD candidature would have undoubtedly been a never ending challenge. I would also like to thank my supervisors, Professor John Gough and Professor David Abramson for their assistance, support and patience; their guidance through the perils of being a PhD student is greatly appreciated. I’m also grateful to Wojtek Goscinski for exposing the VSGuard interface required for my work and, providing unlimited support in an extremely efficient manner. Finally, I would like to thank my colleagues in the Programming Languages and Research Group at the Queensland University of Technology, Diane, Dominic, Gavin, Glen, Greg, Jens, Jiro, Mimi, On, Simon and Richard provided refreshing, and sometimes necessary, distractions that allowed me to keep my sanity in check.
1 Introduction

“... a system always needs to evolve to meet the requirements of the changing world in which it operates.” – Grubb & Takang, 1992 [56].

Software systems often evolve beyond their original operational and hardware requirements. This poses great challenges, both technical and managerial, that may lead to drastic outcomes in the case of failure. During the Gulf War in 1991 for example, an American Patriot Missile defense system failed to track and intercept an incoming Scud missile [54]. The Patriot system classified the incoming Scud missile as a false alarm and took no action - an error that resulted in 28 deaths and over 100 injuries. Four years later on June 4, 1996, after a decade of development costing $7 billion, an unnamed Ariane 5 rocket was launched by the European Space Agency (ESA) [48]. Thirty seconds after lift-off, at an approximate altitude of 3700 meters, the $500 million rocket veered from the intended flight path, broke up and exploded.

These two incidents were the subject of extensive investigations and highlight that software evolution can introduce subtle errors that are difficult to detect for a large number of reasons. Investigation of the first incident revealed that internal calculations became less accurate the longer the system was operational. At the time of the incident, the system had been operational for more than 100 hours and incorrectly calculated that the Scud missile should be 687 meters from its actual location. The Patriot Missile defense system had not previously failed because it was originally designed as a defense system against aircraft and had never been used to provide protection against enemy missiles. The original purpose intended that the system be mobile and operate no more than a few hours at a time, and hence the error in the predicted location was never large enough to misclassify an incoming aircraft.

The Ariane 5 incident, on the other hand, was the result of upgraded hardware. The Inertial Reference System (SRI) failed to transmit the correct data after attempting to convert a 64-bit floating point value to a 16-bit signed integer. The Ariane 5 rocket was equipped with the same Inertial Reference System (SRI) unit as its predecessor, the Ariane 4. However, the initial trajectory of Ariane 5 differs from that of the Ari-
ane 4 and produces a significantly higher horizontal velocity that meant the floating point number was larger than the maximum value that can be represented by a 16-bit signed integer and resulted in a software exception.

These two incidents illustrate that applications constantly evolve and, as a result, are susceptible to subtle errors that can be extremely difficult to detect and locate. The process that involves the initial software development followed by ongoing maintenance releases that accommodate enhancements and corrections to the code, and changes to the operating environment [85] is now widely recognized as software evolution.

According to Lehman [85], as much as 70% of the lifetime cost of software is expended after the software is first released. Another conclusion drawn by Lehman is that if a system is used, it is never finished [83]. Grubb & Takang [56] support this phenomenon by explaining that a system “always needs to evolve to meet requirements of the changing world in which it operates”.

Many software engineering tasks result in software evolution and have the potential to introduce errors. One such task is software maintenance which involves the modification of existing software to correct faults, improve performance or extend existing functionality. After the maintenance task is completed, the software is said to have evolved and a new program version is created. Maintenance consumes a large portion of the system life cycle and overall costs, yet it is one of the least studied areas in software engineering. A survey of empirical research in software maintenance [78] reported that “only 2 percent of empirical studies in software engineering focused on maintenance, despite reports that at least 50 percent of software effort is devoted to this activity [77, 84, 117]”.

Another software engineering task that results in software evolution is the porting of an application from one platform and/or language to another. Porting an application can be a difficult and time consuming process that may be undertaken for a number of reasons. Such reasons can include performance gains, new software technologies, managerial or business reasons, or the introduction of new hardware. The difficulty of the task can range in magnitude from simply re-compiling the application with a
new compiler through to re-writing the entire application. One of the more difficult and time consuming porting tasks is moving an entire application from a serial design to a parallel model, where the application and data is distributed across a number of processes. Regardless of the motivation for porting an application, the task is highly susceptible to subtle errors that are difficult to locate. Examples of common porting problems that can be difficult to isolate are: different data representations, or formats, on the target machine and/or language; precision of floating point numbers that can accumulate errors and produce large discrepancies in computational intensive applications; external libraries/components on the target machine that produce slightly different results on the target machine/language.

When a program is discovered to produce incorrect results, the problem needs to be identified and corrected, a process known as debugging. The developer usually locates the source of an error by observing the behaviour of a program as it executes and comparing it with the expected behaviour. A debugger is a tool used to assist the process of debugging by allowing the execution of a program to be controlled while examining the program’s state. While debuggers are used extensively to identify and correct software faults, they still require the user to have a detailed knowledge of the program in order to determine where the actual behaviour differs from the expected behaviour, and identify the source of a problem.

A large amount of research has been directed at debugging techniques and, in particular, debuggers. As a result, debuggers are continually evolving with new features that assist developers locate program defects. Significant enhancements made to conventional debuggers have been the integration of graphical user interfaces to improve ease of use, data visualization facilities to aid the interpretation of large and complex data structures, process groups to aid the management of many independent threads in parallel machines, and support for parallel and distributed debugging.

Despite the amount of research that has occurred in the area of debugging, little research or support has focused on debugging programs that have undergone software evolution and, in particular, the ability to utilize information from previous program versions. Most existing debuggers are geared toward the development of new programs. By dealing with each program in isolation, they offer only limited support for
the maintenance and evolution of computer programs or for the conversion of software from one machine or language to another.

In 1994, Abramson and Sosic proposed a new technique that exploits information from previous program versions to aid the debugging of applications that have undergone software evolution. The technique, known as Relative Debugging [11, 12, 14, 15], relies on the premise that, in many situations where software evolution takes place, the contents of key data structures remain invariant across releases. This property allows Relative Debugging to utilize information from previous releases to identify and localize faults in new versions of the software.

Relative Debugging enables programmers to locate errors by comparing the contents of key data structures in the new version with the corresponding data structures in an existing version as the two programs are concurrently executed. If the values of two corresponding data structures differ at a point at which they should not, an error may exist and the developer is notified. By observing the divergence of key data structures as the programs are executing, the programmer is able to make informed decisions as to the likely cause of the problem. Relative Debugging is effective in these situations 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. The technique also reduces the need for developers to have a detailed understanding of the program in order to compare the actual program behaviour with the behaviour that is expected.

The concept of Relative Debugging is both language and machine independent. This allows a user to concentrate on the cause of the error rather than the underlying implementation details. The results of several case studies have been reported [11, 12, 132, 133] and show that Relative Debugging is an extremely effective technique that allows users to locate errors in programs that have been modified or ported to another platform and/or language.

Even though the Relative Debugging paradigm is now well established, and has proved to be successful for debugging programs that have evolved, it was clear that a number of aspects should be improved to enhance the existing paradigm and further
assist the process required to debug programs that have undergone software evolution.

Firstly, the Relative Debugging paradigm requires a user to identify corresponding data structures - and locations where these should be compared – within two versions of a program. To do this requires a detailed knowledge of both programs, without which locating either the data structure or comparison points becomes difficult. Up until now the paradigm has not provided direct assistance in this regard; the purpose of our research has been to identify techniques that will overcome these limitations.

The process of identifying data structures and program points in order to locate the source of an error is an iterative and labor intensive process. The user typically identifies a number of data structures and comparison points, executes the two programs, and analyzes the results. If the source of the error is not uncovered or localized, the user will typically identify further data structures and comparison points and re-execute the two programs. This process normally continues, in an iterative fashion, until the user has traced an error back to the origin of the fault. Hence, the second aspect that hinders the existing Relative Debugging paradigm is the large amount of manual effort required.

Our research efforts have been directed at enhancing the currently defined paradigm and addressing the problems mentioned above. The outcome of our research is the discovery of techniques that allow the Relative Debugging paradigm to be significantly enhanced. Specifically, the research has resulted in the following three contributions:

1. *A Systematic Approach to Relative Debugging.* Since the inception of Relative Debugging in 1994, the technique has proven to be extremely useful for debugging programs that have undergone software evolution. From reports and observations, we define a systematic approach to Relative Debugging that allows experienced and inexperienced users to quickly and effectively locate errors using the Relative Debugging paradigm.
2. **Data Flow Browsing for Relative Debugging.** We introduce a *Relative Debugging Data Flow Browser* that assists the user to quickly and effectively identify corresponding data structures, and program points, that will uncover the source of an error. While the technique provides valuable assistance during debugging it also exhibits the following novel features that make it distinguishable from existing data flow browsers:

- **Efficiency:** Any interactive debugging tool needs to be efficient and respond to user queries in a timely fashion. A Relative Debugging Data Flow Browser is no different, especially because a user will typically trace a number of erroneous variables in order to locate the source of a problem.

- **Multi-lingual:** An important application of Relative Debugging is the debugging of programs that have been ported from one language to another. To ensure that a Relative Debugging Data Flow Browser remains consistent with this use, it must be capable of processing many languages.

- **Usability:** To ensure that a Relative Debugging Data Flow Browser remains an effective tool that allows users to quickly and efficiently trace the data flow of a program, it must present the results to the user in a clear and concise manner. Hence, a Relative Debugging Data Flow Browser must be embedded in an IDE, allowing a user to navigate complex programs quickly. This also allows users to trace data definitions whilst executing the program under debugger control, providing a unique and powerful platform for locating errors.
3. **Automatic Relative Debugging.** We have pioneered an approach to automate the process of Relative Debugging. To do so we have leveraged a number of techniques, which have led to the following:

- *An approach that allows matching data structures and program locations to be automatically identified such that errors can be traced back to their origin.*

- *An approach that allows suitable data structures and program locations to be identified when the programs differ and matching data structures cannot be identified when tracing an error back through a program.*

- *An approach that allows variables that have been renamed to be identified and automatically matched for comparison.*

These contributions have enhanced the Relative Debugging paradigm and allow errors to be localized with little human interaction. Minimizing the user’s involvement reduces the cost of debugging programs that have undergone software evolution. Automating the Relative Debugging paradigm reduces the need for users to have a detailed knowledge of the programs under consideration, thereby improving productivity and has a significant impact on current debugging practices.

The remainder of this thesis presents our research contributions and is organized as follows:

Chapter 2 provides an overview of software evolution and the inherent costs that motivate the ongoing research in this area. The lack of debugging support for programs that have undergone software evolution is highlighted and provides the motivation for the research performed and presented in this thesis.

Chapter 3 provides a detailed description of Relative Debugging and provides the background information that motivated our research. We then draw on our experiences and observations to define a systematic approach to Relative Debugging that
allows both experienced and inexperienced users to quickly and effectively locate errors. The approach also provides an important framework that allows us to achieve our other two contributions, discussed in the following two chapters.

Chapter 4 introduces data flow browsing for Relative Debugging, a technique that provides invaluable support when using Relative Debugging to locate errors in programs that have undergone software evolution. The chapter explains why data flow browsing is extremely important when using Relative Debugging and how it differs from other data flow browsing techniques. A practical design for a Relative Debugging Data Flow Browser is provided, followed by a discussion of a supporting implementation. The chapter concludes with a case study that demonstrates the application of a Relative Debugging Data Flow Browser and highlights the productivity gained.

Chapter 5 presents the main contribution of our research - novel techniques that allow the Relative Debugging paradigm to be automated. The techniques that allow Relative Debugging to be automated are discussed, followed by the design and implementation of an automatic Relative Debugger.

Chapter 6 presents three case studies that demonstrate the application of automatic Relative Debugging to localize faults with little human intervention. The case studies have been organized to clearly highlight the individual contributions of the research. The case studies demonstrate that automatic Relative Debugging is effective and significantly improves productivity.

Chapter 7 summarizes the research detailed in this thesis and provides some future directions that could further improve the Relative Debugging paradigm.
2 Background and Motivation

“... as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.” - Edsger Dijkstra, 1972 [42].

The software crisis was the term given in 1968 [8] to describe the rapid increase in the number of systems that did not meet client expectations and/or exceeded the allocated development time and budget. To overcome this crisis it was accepted that the creation of software should be treated as a genuine engineering activity and demanded a structured and visible development process.

The first model describing the software development process was proposed and enthusiastically accepted in 1970 [113]. It soon became apparent however, that the model was not suited for all software development projects. As a result, a number of models have evolved and are in widespread use today [122].

The models describe the activities, or phases, that are performed during the life of a software system. Although the tasks performed during each phase, or the order in which the phases are performed may differ, a number of models incorporate common phases. These common phases are requirements analysis, design, implementation, testing and operation and maintenance.

Research and development has occurred in all phases of the software development life cycle. The least studied area is the operation and maintenance phase although it has been identified as the most expensive. Kemerer and Slaughter [78] conducted a survey of empirical research in software maintenance and reported that “only 2 percent of empirical studies in software engineering focused on maintenance, despite reports that at least 50 percent of software effort is devoted to this activity [77, 84, 117]”.

27
The remainder of this chapter provides an overview of the software maintenance phase and the inherent costs that motivate the ongoing research in this area. The focus is then shifted to the lack of support for debugging activities within this phase, and provides the motivation for the research performed and detailed in this thesis.

2.1 Software Maintenance and Evolution

Software maintenance and evolution are closely related. Software maintenance is defined in the IEEE Standard 1219 [70] as "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." In line with this definition, Martin and McClure [94] have identified the following reasons for performing maintenance:

- Correct errors.
- Correct design flaws.
- Interface with other systems.
- Make enhancements.
- Make necessary changes to the system.
- Make changes in files or databases.
- Improve the design.
- Convert programs so that different hardware, software, system features, and telecommunications facilities can be used.

Another important reason identified by Pigoski [107] is to:

- Prevent problems or to improve performance.

The lack of a common understanding and standard definitions in software maintenance research was first identified by Swanson in 1976 [129]. To eliminate this problem, and provide a common framework in which empirical research could be conducted, he proposed 3 maintenance categories [129]. These categories were widely accepted and used in numerous research efforts. In 1993, IEEE further refined these categories, and added a fourth [70]. In general terms, these categories are:

- **Perfective Maintenance** – software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.
- **Corrective Maintenance** – software maintenance performed to correct faults in hardware or software.
- **Adaptive Maintenance** – software maintenance performed to make a computer program usable in a changed environment.
- **Preventative Maintenance** - software maintenance to update the software in order to improve its future maintainability, without changing its current functionality.

In each of these cases the result of the maintenance activity is a new program version. This continuous program growth, through the initial development and ongoing maintenance, is referred to as software evolution.

Software evolution lacks a standard definition, and hence, the distinction between software maintenance and software evolution is not clear. Belady and Lehman suggest that software evolution is “the dynamic behaviour of programming systems as they are maintained and enhanced over their life times” [31]. Ramil and Lehman similarly define software evolution as “a continuing process that encompasses all activities, such as enhancement, adaptation or fixing, that occur after the delivery of the first operational release to the users” [110]. In either case, the definition clearly encompasses the four IEEE maintenance categories, and therefore, we can safely presume that software evolution subsumes maintenance activities. In addition, an implicit property of the two definitions is that the activity results in a new program version.

### 2.1.1 The Cost of Evolution

The observation that systems continually evolve was first suggested by Belady and Lehman in the early seventies [31]. They identified five common properties for software evolution planning and management during an empirical study of the OS/360 operating system [31]. After further studies [110] these rules have been accepted as useful guides for managing the software life cycle.

While the study was not aimed at identifying or estimating the cost of software evolution the following observations, or so-called laws, imply that systems evolution, and the associated costs, are unavoidable:
Law of Continuing Change. A program that is used must be continually adapted else it becomes progressively less satisfactory.

Law of Increasing Complexity. As a program is evolved its complexity increases unless work is done to reduce it.

Law of Continuing Growth. Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

Ramil and Lehman [110] discovered that little work exists on identifying the cost of software evolution and consequently started exploring cost estimation models using the extensive data retrieved from Lehman’s earlier work.

Conversely, a number of studies have been conducted to investigate the cost of software maintenance. As an example, an extensively cited study was performed by Leintz and Swanson to investigate the cost of software maintenance [87]. The empirical research, conducted over 3 years in the late 1970s, collected data from 487 software organizations. The results showed that programmers contribute approximately half their effort on maintaining a developed system. The survey also revealed that 50% of total maintenance costs can be attributed to perfective maintenance, 25% for adaptive maintenance, while only 21% of the total costs are attributed to corrective maintenance, and 4% for preventative maintenance.

Although the study by Swanon and Lientz is dated, several studies have validated their findings. The results conclusively reveal that maintenance consumes a large portion of the system life cycle and overall cost, as illustrated in Figure 2-1
Figure 2-1: Cost of software maintenance as reported by several surveys.
2.1.2 The Evolutionary Process and Supporting Tools

Software maintenance and evolution is becoming recognized as an important area of software development and tools are slowly emerging. However, “there is still a need for better tools, techniques and methods and, perhaps most importantly, better education and training” [122].

The ISO/IEC 9126 (1991) standard defines the following four steps that are performed when a system is modified and undergoes evolution [71]:

1. Analysis.
2. Change.
3. Restabilize.
4. Test.

The analysis step requires programmers to gain an understanding of the program in order to locate where changes must be applied and, determine the impact of the changes. This task is typically referred to as program comprehension and has been identified as a costly task that consumes between 50 and 90 percent of the overall maintenance effort [50, 126]. As a result, significant research has been devoted to the development of tools and methodologies that aid the task of program comprehension [29, 96, 120]. A full list of program comprehension approaches is presented in [114] and highlights that program comprehension has been the subject of significant research and development. The list includes:

- Textual analysis, lexical and syntactic analysis.
- Data and control flow analysis.
- Program dependence graphs.
- Slicing.
- Cliché recognition.
- Abstract interpretation.
- Dynamic analysis.
- Partial evaluation.
If the testing phase unveils an error the user must debug the program. Most existing debuggers however, are geared towards the development of new programs, and offer only limited support for programs that have undergone software evolution.

### 2.1.3 Current Debugging Support for Software Evolution

In many situations, when a program is extended or ported to another platform and/or language, large portions of the modified program should produce the same results as the existing version. A common approach to ensure that these program portions produce identical results is to manually inspect and compare the modified code with the original version. This approach is manually intensive and susceptible to human error i.e., errors are easily overlooked.

A more intuitive and powerful approach is to compare the execution of the development version with the execution of the original version. This is commonly achieved by displaying, at regular intervals, the state of two programs and comparing the differences. This approach becomes tedious because source code changes that capture and output the program’s state are typically required. Comparing the output can also be a difficult task if the data types are represented differently by the two programs e.g., floating point numbers. In this case, a simple text based comparison utility may report a large number of false positives and impede the practicality and efficiency of this approach. Another approach to ensure that the development version is producing the expected results is to compare the execution with that of the original version. This may be achieved by simultaneously executing the two programs under the control of two separate debuggers. This approach also requires significant human effort which is extremely tedious and susceptible to errors.

There are few tools that currently assist the developer to locate the erroneous code responsible for producing the contrasting results. In particular, most existing debuggers are geared toward the development of new programs and provide limited support for the maintenance and evolution of computer programs or the conversion of software from one machine or language to another.
The following two debuggers, however, provide the functionality to debug programs that have undergone software evolution:

- Guard [11, 15].
- p2d2 [65].

### 2.1.3.1 Guard: A Relative Debugger

Relative Debugging is a paradigm [12] that assists users to locate errors in programs that have been corrected or enhanced. In particular, the contents of key data structures in the development version are compared with the contents of the corresponding data structures in an existing version as the two programs execute. If the values of two corresponding data structures differ at points in which they should not, an error may exist and the developer is notified.

A number of Relative Debuggers have been developed by Abramson and successfully used for a number of years. The implementations have demonstrated that the Relative Debugging paradigm is extremely effective and significantly improves productivity [11, 12, 132, 133] when debugging programs that have evolved.

The first implementation of a Relative Debugger, named Guard95, was developed by Abramson and Sosic in 1995 at Griffith University in Brisbane, Australia [13]. This proof-of-concept implementation provided conventional debugger commands as well as a new set of commands that facilitated the comparison of a new program version with an existing version. Guard95 only provided support for the comparison of variables and arrays of elementary types.

To allow Guard to function in a distributed and heterogeneous environment, Guard95 required the use of a portable debugging interface, called Dynascope [123, 124]. Dynascope defined a collection of interfaces that provide support for execution control, state access, breakpoints, tracing, and dynamic linking. The Dynascope interfaces have been implemented and successfully used on a number of architectures, including Sun, Next, and Silicon Graphics, and IBM RS600 machines.
Since the first implementation of Guard, Relative Debugging and Guard have been the subject of significant research. This research has resulted in the following innovations:

- **A portable debugging interface.** The use of Dynascope was replaced by GNU’s GDB [125] in order to avoid the cost of developing and maintaining the low level debugging functions on each target machine. The use of GNU’s GDB provided a freely available, well-tested debugging framework, and the ability to deliver Relative Debugging on over 20 different architectures.

- **Parallel Relative Debugging.** A significant addition to Guard is the ability to apply Relative Debugging techniques in situations where a serial program has been ported to a parallel program, or a parallel program has undergone evolutionary software development [16, 17]. This functionality was introduced by Watson and Abramson in 1996 and is extremely useful since errors are often introduced when sequential program are ported to a parallel model.

- **An architecture independent format.** Relative Debugging requires data from two programs to be compared. Data may be represented in different formats depending on the architecture of the underlying machine, a problem that Guard95 did not address. This problem was addressed by introducing an architecture independent format (AIF) to overcome the problem with managing different data representation [16, 17]. AIF was designed to achieve true architecture independence for arbitrary types. The key components of the AIF system are:
  - A format for representing data in an architecture independent manner.
  - Routines for converting to and from native formats.
  - Routines for performing arithmetic, logical and comparison operations on AIF.
  - Routines for accessing components of structures data types.
  - Routines for performing I/O on AIF data.
• Data structures. Guard was originally designed to compare the contents of static structures like scalars, simple structures and arrays. Guard has since been extended to handle the comparison of dynamically allocated structures [10], such as linked lists.

• A transformation algebra. A key component of parallel Relative Debugging is the ability to compare data structures between serial and parallel codes. It is likely that the data from the sequential program will be distributed across more than one process in the ported parallel program. A special syntax, derived from a transformational algebra, has been developed by Watson to model the different types of data transformations that may occur [133, 134]. This allows users to compare data structures between the sequential and parallel versions without concern for the different organizations of the data in the two programs. The syntax currently allows the following four types of transformations to be defined: Data Parallel Decomposition, Shape Transformations, Index Permutation, Temporal Displacement.

The early versions of Guard were command line driven, and thus required the user to maintain separate windows onto the two source programs and the debugger. Recently, Guard has been integrated into a number of different interactive development environments by Abramson [18], namely:

• VSGuard is an implementation of Relative Debugging for the Microsoft Visual Studio .NET ® environment [98]. It supports all .NET Framework languages, as well as legacy Microsoft languages like Visual Basic 6.0 and Visual C++.

• Eclipse Guard is a version of Guard integrated into the IBM Eclipse platform [2]. Eclipse Guard leverages the flexibility and extensibility of Eclipse. It currently works with Java and C/C++, however, this can be extended as new language components are produced.
These IDEs offer significant advantages to Guard as they allow the user to maintain two separate source code views concurrently, and side-by-side in the environment. Further, identifying key structures and comparison points within the two programs is easier, because it is possible to point and click on the variables and lines of interest rather than needing to find the line numbers and manually entering this information on a command line.

2.1.3.2 p2d2: A Portable Distributed Debugger

Hood and Jost [66] describe a system, developed at NASA’s Ames, that simplifies the process of debugging programs produced by computer-aided parallelization tools. The system uses Relative Debugging techniques to compare the execution of the serial code with the corresponding parallel code produced by the parallelization tool. If the original serial code is correct, errors due to parallelization will be highlighted by showing where computations begin to differ.

The system is not as flexible as the Guard Relative Debugger because the system is tightly integrated and highly dependent on other applications in order to provide the framework for parallel Relative Debugging. The required applications that support the framework are:

- A set of tools developed by the University of Greenwich that support the transformation of serial code to the parallel equivalent. A necessary component produced by CAPTools is a database that contains the details of the data structure distribution and transformations that occurred when the parallel code was produced [38]. This database is required for p2d2 to perform Relative Debugging.
- A dynamic instrumentation (Dyninst) tool used to dynamically insert code that executes the required data structures comparisons at runtime [59].
- A portable parallel/distributed debugger developed at NASA’s Ames [65].

Unlike Guard, p2d2 is restricted to debugging programs that have been generated using the parallelization tool described above. Furthermore, the granularity of comparisons is restricted by limitations in the dynamic instrumentation tool, Dyninst. Comparison routines are dynamically inserted on entry and exit of routines that modify the distributed arrays of interest. The system uses the database produced by the
parallelization tool to determine how the arrays are distributed across multiple processors in order to correctly place the corresponding comparison functions in both the serial and parallel versions.

### 2.2 Automatic Debugging

In 1991, Agrawal stated that “automatic debugging has been investigated but never really achieved” [19]. A review in 1988 had already noted that “most automatic debugging tools were currently only prototypes and did not scale easily to large systems” [43], mainly due to the difficulty involved with capturing the programmer’s intuition.

Since the early eighties, a number of research projects and experiments that adopt novel approaches for automatic debugging have been conducted and have provided promising results. These approaches rely on the correctness of earlier versions of a program to assist the programmer’s understanding of the development system and to help with the isolation of faults in development versions. The approaches attempt either to localize faults, or to reduce the search area to a minimal one that allows the errors to be reproduced.

#### 2.2.1 Shapiro’s Interactive Diagnosis Algorithm

Shapiro proposed *diagnostic algorithms* [119] to identify procedures, or functions, that behave incorrectly. Shapiro proposed an approach known as *algorithmic debugging*, which is a semi-automatic technique where the user is required to answer questions concerning the intermediate results of procedures within the program.

Shapiro explains that the “approach is geared towards languages in which the basic computation mechanism is a procedure (or function) call, but is insensitive to the inner workings of procedures. The diagnosis algorithms abstract away all the details of the computation, except the procedure calls performed, their inputs, and their outputs.” [119].

The problem with the basic algorithm is the potentially large number of queries that the user may be asked. To reduce the number of queries that the user may be asked, Shapiro suggests the following techniques:
• Accumulate a database of answers to previous queries and use them to answer repeated queries.

• Supply assertions and constraints on the input/output behaviour, invoke the diagnosis algorithms whenever they are violated, and use them to answer diagnosis queries whenever they apply.

• Use a previous version of the program that is known to work to answer queries.

Shapiro implemented a Prolog implementation of these algorithms, capable of diagnosing pure Prolog programs.

2.2.2 vpoiso

In 1994 Whalley presented a tool, vpoiso [136], that automatically localizes errors in the vpo compiler system. The tool isolates optimization errors by determining the first transformation that causes the compiled program to produce incorrect results. Further, vpoiso determines a minimal input which causes the compiled program to produce the incorrect result. Although the tool cannot isolate non-optimizing errors, the minimal input narrows the scope of the problem thereby assisting the developer to localize the error.

To isolate an erroneous optimization transformation, vpoiso systematically enables each optimization in turn and compares the results of the compiled program with the expected results provided by a compiled non-optimized version. This process continues until the first transformation that causes the program to produce incorrect results is identified.

Unlike optimizing transformations, every non-optimizing transformation must be applied to successfully compile a program. Therefore, unlike the process to locate optimization errors, non-optimizing transformations cannot be systematically disabled to isolate an error. For this reason, vpoiso does not identify erroneous non-optimizing transformations, but rather determines the minimal input program that causes the compiler to produce an incorrect program. To achieve this, vpoiso uses a working compiler and systematically replaces functions in the program produced by the erroneous compiler with the corresponding functions in the program compiled
with the correct compiler. This process continues until a program that produces the error, containing the minimal number of functions, is discovered.

This technique assumes that the optimization phases are totally independent in order to identify the erroneous transformation. Hence, the erroneous transformation cannot be isolated if the error only occurs when more than one optimization is enabled. Given that the error is identified by comparing the output of the compilation, a further requirement is that the error is independent of the input. In particular, the error that occurs when one function is compiled cannot be undone by compiling a subsequent function.

2.2.3 DynaDiff

DynaDiff [112] is a testing and debugging tool that uses path profiling to isolate potential faults. The tool runs a program twice, with different inputs, and compares the executed control path. The user is informed that an error may exist if an alternate control path is taken or if a control path is executed a different number of times.

DynaDiff was originally used to detect errors introduced by the Year 2000 Problem [112] (also known as the Y2K Problem). Problems can be attributed to date-dependent computations by supplying inputs that keep all factors constant except the way dates are used in the program. Hence, DynaDiff identified potential problems by determining if a different control path was executed when the date was set prior to the year 2000 than the control path taken when the date was set after the year 2000.

The technique is also applicable to other problem domains provided that the different inputs should result in the program executing the same control paths.

2.2.4 Delta Debugging

*Delta Debugging*, recently proposed by Zeller [37, 139-143], is a novel and powerful technique that automatically isolates the cause of an error. The technique can be used on any artifact, or circumstance, that influences the execution of a program. Example circumstances are program input, user interaction, and the actual program code.
The Delta Debugging algorithm uses systematic testing, similar to vpoiso, to automatically isolate failure-inducing circumstances. A failure-inducing circumstance is a subset of the original circumstance that contains the minimal number of elements that cause the original error to re-occur. Each element in the failure-inducing circumstance must be present in order for the error to occur - removing a single element eliminates the occurrence of the failure. Removing the elements that do not influence or impact the failure allows the user to concentrate on the cause of the error.

To identify the failure-inducing circumstances Delta Debugging systematically removes elements from the original circumstance and re-executes the program to determine if the error remains. This process continues in an iterative fashion until the original circumstance is decomposed to the subset of minimal circumstances that reproduce the original error.

Removing elements in an arbitrary manner and testing each combination can be extremely inefficient. For instance, to simplify a circumstance, or input, of $n$ characters requires at least $n$ iterations i.e., removing each character in turn and performing the test with the modified input. Another problem associated with removing elements in an arbitrary fashion is that a circumstance may not be valid after removal of an element. For example, assuming the user’s interaction with a program had been recorded, it would not make sense to remove the user’s selection of a main menu option and leave selection of the submenu option. The resulting input would not be valid, and could not be used as a valid test case, because the sub menu option cannot be selected without first selecting the main menu option. Such a test case is referred to as inconsistent.

To ensure that Delta Debugging remains efficient and practical it has been developed using a divide and conquer algorithm. The algorithm proceeds by partitioning and testing two coarse partitions. Partitions that pass need no further investigation and can be discarded. Partitions that fail are subjected to further partitioning and testing. To further improve efficiency, but more importantly, avoid testing inconsistent circumstances, Delta Debugging may also use knowledge from the problem domain, such as:

- *Process information* such as common change dates or sources.
• *Location information* such as affected files or directory.
• *Configuration management* such as version control information.
• *Lexical information* such as common referencing of identifiers.
• *Syntactic information* such as language syntax.
• *Semantic information* such as data or control flow.

The efficiency can be improved even further if a passing test case is available. The existence of a passing test case allows the technique to identify and process the changes between the passing and failing test case in order to isolate the failure-inducing differences.

Several case studies that show Delta Debugging to be an effective technique to localize faults have been performed. [37, 140] details the use of Delta Debugging to isolate the problem that caused GNU DDD to function incorrectly when used with an updated version of the GNU GBD [125] debugger. GNU DDD [144] is a graphical front-end for UNIX command-line debuggers such as GNU GDB. Delta Debugging was successfully used to isolate the GDB modification, consisting of 178,000 changed lines, that caused DDD to function incorrectly. An initial attack on the problem, using no knowledge about the changes or a previous program version, was performed and required 470 tests that took 48 hours. Another test run, which used location and syntactic criteria, required only 97 tests and 4 hours.

Another case study is detailed in [141] and highlights the ability of Delta Debugging to isolate failure-inducing circumstances. Delta Debugging minimized the sequence of user actions that causes the Mozilla [6] web browser to terminate with a segmentation fault. After the original 95 user actions were reduced to just 3, Delta Debugging was further used to reduce the HTML source, using the 3 user actions, from 896 lines to just 1 problem line that caused the error to re-occur.

Delta Debugging may also be used to isolate faults in computer programs. In this case, the actual computer program is considered the circumstance, or input, that is processed by the Delta Debugging technique. The algorithm attempts to locate the code that influences the failure. The case study in [141] for example, used Delta
Debugging to isolate an error that caused the GNU GCC compiler [5] to terminate unexpectedly (a.k.a. crash). In this case study, the program that causes the compiler to crash was reduced from 755 characters to a small program containing only one function comprised of 77 characters.

The Delta Debugging technique has since been applied to actually isolate the fault within an erroneous program by systematically narrowing the state difference, i.e. variables and their values, between a passing run and a failing run [142]. This approach interrupts execution of the passing and failing runs at predetermined locations. The state of each program is then retrieved and the set of variables that differ is identified. This set may contain variables that do not cause the error to occur. To eliminate such variables the set is subjected to Delta Debugging. The resulting set represents the difference in the program state which eventually causes the failure. These isolated variables constitute the cause-effect chain that leads the developer from the root cause to the failure.

### 2.2.5 WHITHER

A similar approach to Delta Debugging has been adopted by Renieris and Reiss [111], but they attempt to address the assumption that similar inputs result in similar runs. To overcome this limitation they select a successful run, from a repository of program runs, that most resembles the faulty run. The selected successful run is then executed and compared with the execution of the faulty run. The difference of these runs is then reported to the user to assist with locating the source of the error.

Two techniques that attempt to select the most suitable run for comparison have been implemented in a prototype tool called WHITHER, and evaluated with varying success. A further comparison method using potential invariants [47] is detailed in [108]. This approach augments the program with simple comparisons that test arbitrary conditions that have been discovered true for a number of successful runs.

### 2.3 Automatic Relative Debugging

Relative Debugging is different from the approaches mentioned above. In particular, the p2d2 system simplifies the process of debugging programs produced by computer aided parallelization tools. However, the framework that provides automatic debug-
ging of parallelized code relies heavily on the integrated applications. Unlike Guard, p2d2 is restricted to debugging parallel programs that have been generated using the associated parallelization tool. The tool records information regarding the parallelization of the code and distribution of variables and arrays across the multiple processes. It is unable to debug programs that have been built outside this framework, and in general, is unable to debug serial programs.

The obvious difference between Relative Debugging and the remaining approaches is that Relative Debugging processes two programs with the same input, as opposed to the one program with two different inputs. This difference allows Relative Debugging to locate bugs in programs where no successful run exists.

This difference also eliminates the assumption imposed by Delta Debugging that similar inputs result in similar runs. WHITHER attempts to address this assumption by selecting a successful run, from a repository of program runs, that most resembles the faulty run. The comparison approach using potential invariants, adopted by WHITHER, is similar to Relative Debugging because it compares a variable from the faulty run with the expected value in the successful run. However, the approach compares arbitrary variables based on the successful runs and assumes that the variables in the successful run should be the same for all runs in the faulty run. Furthermore, the compared variables may not actually influence the faulty program run. Conversely, the approach does not consider situations where the compared variables should actually differ between the two program runs.

The research described in the remainder of this thesis has enhanced the Relative Debugging paradigm by automating the process of Relative Debugging. Relative Debugging requires users to identify the data structures within the two programs that should exhibit equivalent values during execution. The corresponding data structures and the program points in which they should exhibit equivalent values during program execution are automatically identified. Minimizing the user’s involvement reduces the cost of porting and converting software from one machine or language to another and has a significant impact on current practices in software development. By automating the process of Relative Debugging, the need for the user to have a detailed knowledge of the two programs is reduced.
3 A Systematic Approach to Relative Debugging

“... a good methodology should lend itself to automation.” - Gillies, 1992 [55].

Software evolution is now widely recognized as a process that involves the initial software development, followed by ongoing maintenance releases that accommodate enhancements and corrections to the code, and changes to the operating environment [85]. In many situations where software evolution takes place, large portions, if not all, of the modified program should produce the same results as the existing version. This is commonly the case when software is modified to remove errors, enhanced with new functionality, or ported from one platform and/or language to another. When a modified version produces results that differ from the existing version, the developer will normally locate and correct the fault by isolating the code that is producing conflicting values.

There are currently few tools that assist the developer to locate the erroneous code responsible for producing the conflicting results. In particular, most existing debuggers are geared toward the development of new programs and provide limited support for the maintenance and evolution of computer programs, or the conversion of software from one machine and/or language to another.

Abramson and Sosic proposed Relative Debugging [11, 12, 14, 15] in 1994 to assist users locate errors in programs that have been corrected or enhanced. Relative Debugging enables programmers to locate errors by comparing the developed (evolved) program with the original (existing) program as they are concurrently executed. Potential errors are highlighted by comparing the execution of the two programs and reporting the divergences that occur between corresponding data structures.

Relative Debugging has been the subject of significant research since its conception in 1994. Research has resulted in the following innovations [16, 17, 132]:

- Portable debugging interface.
- Parallel Relative Debugging.
This thesis presents the latest research that we have conducted in the area of Relative Debugging. The main contributions of our research are detailed in the following three chapters, as follows:

- A Systematic Approach to Relative Debugging (this chapter).
- Data flow browsing for Relative Debugging (Chapter 4).
- Automatic Relative Debugging (Chapter 5)

This chapter begins with a detailed discussion of the current Relative Debugging paradigm. We then present a systematic approach to Relative Debugging. This approach provides the steps a user, experienced or inexperienced, may follow to quickly and effectively apply Relative Debugging to locate errors in software that has undergone software evolution. The approach is also important because it provides the framework that allows us to achieve our other two contributions, as discussed in the following two chapters.

### 3.1 The Relative Debugging Paradigm

As discussed and listed in Chapter 2, programs evolve for a large number of reasons. In many situations large portions of the modified program should produce the same results as the existing version. For instance:

- If a program is enhanced with new functionality, the existing functionality should remain the same.
- If a program is modified to remove an error, the existing unrelated functions should produce the same results. That is, the removal of an error should not introduce new errors.
- If a program is ported from one platform and/or language to another, the functionality in the ported program should remain the same.

In these cases, it is common that the unaffected functionality, or program code in the development version, will contain data structures that should contain the same values
as the original version when the two programs are run with similar inputs.

Relative Debugging [11, 12, 14, 15] exploits the information available in previous versions to allow the contents of key data structures to be compared. This allows the user to locate errors by identifying data structures that contain different values when the two programs are executed.

The concept of Relative Debugging is both language and machine independent. This allows a user to concentrate on the cause of the error rather than the underlying implementation details. The results of several case studies have been reported [11, 14, 15, 133, 134] and show that Relative Debugging is an extremely effective technique that allows users to locate errors in programs that have been modified or ported to another platform and/or language.

Figure 3-1 for example, illustrates the Relative Debugging paradigm. The program on the left depicts the original program, written in VB, containing a function that computes the factorial for a given number. The program on the right represents the later development version which has been rewritten in C++ and, uses a different algorithm to compute the factorial. Despite these differences, the two programs should compute the same factorial value for the given number. Hence, Relative Debugging may be used to compare the factorial value computed by each programs. If the comparison fails, the error can be attributed to the modified factorial function in the development version. In this case, the user may use traditional debugging techniques or continue to apply Relative Debugging to localize the source of the error even further.
Function fact(ByVal n As Integer)
Dim factVal = 1
Dim i = 2
While (i <= n)
    factVal = factVal * i
    i = i + 1
End While
Return factVal
End Function

public int factorial(int n)
{
    int fact;
    if (n == 0) {
        fact = 1;
    }
    else {
        fact = n;
        n = n - 1;
        while (n > 0) {
            fact = fact * n;
            n = n - 1;
        }
    }
    return fact;
}
Relative Debugging requires the user to identify the corresponding data structures and program points at which comparison should be made within the two programs when they are concurrently executed. The choice of data structures and program points must be determined by the user based on some knowledge of the two programs. It is not necessary to test all data structures, but only those which will help uncover the source of the error. The user provides the comparison details by defining one of the following two types of assertions:

- Imperative Assertions.
- Declarative Assertions.

### 3.1.1 Imperative Assertions

An imperative assertion allows the user to manually invoke the comparison of key data structures when the two programs under consideration are suspended. Typically, the user creates a number of breakpoints, at key locations, in the two programs before they are simultaneously executed. When the breakpoints are reached, and the programs are subsequently suspended, the user can issue a `compare` command to compare the contents of key data structures. For example, the following `compare` command compares the variable `factVal` in the program `orig`, with the variable `fact` in the program `dev`.

```
compare $orig::factVal = $dev::fact
```

The user may perform a number of comparisons and gain insight into the error by identifying variables that contain different values across the two programs. To identify where the two programs first begin to differ, and isolate the source of the error, the user can define additional breakpoints, resume or restart the two programs, and continue to compare the contents of key data structures.

This process can become tedious and error prone if the user needs to iterate through this process a number of times. The process also becomes laborious if the user wishes to compare the results of a computation that occurs within a loop. In this case, the user must manually invoke the comparison each time an iteration is performed and the programs are suspended.
3.1.2 Declarative Assertions

A more flexible, and less tedious method for defining the comparison details of corresponding data structures is specified by a user-supplied declarative assertion. The declarative approach allows the user to assert that two data structures should contain the same values at particular locations in the two programs.

Declarative assertions are defined before the programs are executed and state that a data structure at a certain line in the original program should contain the same value as the corresponding data structure, at a specific line, in the development program.

Declarative assertions are defined by the assert command. The following assertion statement, for example, states that the variable factVal, on line 114, in the original program should be compared with the variable fact, on line 121, in the development program.

\[
\text{assert } \text{orig::factVal@114} = \text{dev::fact@121}
\]

The additional line information provided with declarative assertions allows the creation of breakpoints and the comparison of variables to be automated by the Relative Debugger. In particular, the Relative Debugger creates a breakpoint in both the original and development versions, for each assertion, at the specified lines. When these breakpoints are reached during execution of the two programs, the Relative Debugger automatically compares the variables specified by the assertion. If the comparison does not detect an error, the programs are automatically resumed without any user interaction.

Declarative assertions provide a convenient mechanism for stating a set of conditions that must be satisfied if the development version is to be deemed correct. Hence, not only are they useful for isolating the source of an error, they can also be used to automatically test a new program version against a previous version.

Declarative assertions are also effective if the user wishes to compare the results of a computation that occurs within a loop. Unlike the imperative approach, the compari-
son is automatically performed each time the loop is iterated. As the user is not involved until an error is detected, little user interaction is required to actually detect the presence of an error.

### 3.1.3 A Systematic Approach to Relative Debugging

Once the user has defined the assertions, the programs are concurrently executed and the values of corresponding data structures are compared at the specified locations. If a comparison fails, execution is halted and the difference is reported to the user. In the case of an error, the user normally repeats the process of program comparison, with additional or refined assertions, in order to identify the point where values first begin to differ.

To identify the point where values first begin to differ, the user typically formulates additional assertions based on the variables used in the computation of the erroneous value. The methodology for deciding where to place assertions is based on following the data and control flow of the two programs. For example, if the output of a computation is reported to be incorrect, the user would define additional assertions to follow the inputs to the faulty computation – typically by finding the definition points of the variables used in the computation. This process usually continues by a process of refinement until the faulty section of code is localized after which traditional debugging techniques can be used to correct the error.

This systematic approach allows users to locate an error by iteratively defining assertions until the erroneous region is small enough to inspect manually. The technique is extremely efficient because the amount of code that needs to be considered is dramatically reduced after each refinement step. Hence, even in large programs, the user can localize the error to a small region of code with a limited number of refinement iterations.

The abstract notion of successively refining assertions on a pair of programs is already at the heart of the Relative Debugging method. By defining a **systematic approach** we refine this underlying principle to a structured and machine guided process.
3.1.3.1 Example

To demonstrate the methodology, consider the two program versions in Figure 3-2. Although the two versions should produce the same result, version 2 computes and displays a different result to version 1. To locate the origin of the error, the user identifies the output variable, \( z \), and locates the definition(s) that may compute the erroneous value. In this case, the value of \( z \) may be computed by one of two definitions. To determine which computation is producing the incorrect value the user creates an assertion for each definition. These assertions are represented by assertions 1.1 and 1.2.

After running the two programs, the user is notified that assertion 1.1 has performed a comparison and reveals that the value of \( z \) differs between the two versions. Assertion 1.2, on the other hand, was not executed. This outcome allows the user to discard the definition tested by assertion 1.2 and concentrate on the erroneous computation tested by assertion 1.1. The definition of \( z \), tested by assertion 1.1, uses two variables, \( x \) and \( y \). Accordingly, the user creates assertions 2.1 and 2.2 for the variables \( x \) and \( y \) respectively. Once again, the programs are run, and the results reveal that the variable \( y \) contains the same value in both programs. The definition for \( x \), on the other hand, shows that the value computed in program 2 differs from that in program 1. The user repeats the process, creating assertions 3.1 and 3.2, to follow the inputs to the erroneous computation of \( x \) tested by assertion 2.1. After running the programs again, assertion 3.2 fails and identifies the source of the error because the definition of \( b \) in this case does not depend on any further variables.

This example has illustrated the methodology that a user should follow in order to efficiently localize the origin of a fault. The methodology was illustrated by considering each step in isolation and creating the minimal number of assertions. In practice, the user may generate a number of assertions that will reduce the suspect region during one refinement step even further. Hence, in the above example, the user could define all the assertions and locate the origin of the error within one iteration.
3.2 Enhancing the Relative Debugging Paradigm

To create assertions in a quick and effective manner requires that the user has a detailed knowledge of the two programs under consideration. Without a detailed knowledge, the user must rely heavily on the control and data flow information of the two programs. Specifically, in order to determine why a variable used in an expression is incorrect it is necessary to find the variable’s definitions in the code. In complex multi-file programs however, there may be multiple definitions that are sparsely scattered through the program. Locating each definition can be a complex and time consuming task. The next chapter introduces a Relative Debugging Data Flow Browser that allows the user to easily discover and traverse variable definitions and effectively define assertions.

The refinement process of building assertions that trace the flow of errors and locate faulty sections of code is well defined and allows the methodology to be automated. We have automated the methodology that is used when constructing assertions. In a
manner very similar to the manual approach, assertions that trace the flow of errors back through a program are systematically generated in order to locate the source of an error.

The technique allows faults to be localized with little human intervention and reduces the need for users to have a detailed knowledge of the two programs under consideration. The system accepts two versions and identifies where the programs first begin to differ during execution. Such automation improves the current methods for debugging programs that have undergone software evolution and improves productivity.

The technique is very effective when the programs are minor variants of each other. It allows automatic debugging of programs when they are simply recompiled on another system and identifies errors introduced by the use of different runtime libraries or other external components. In situations where the development version has been modified such that matching data structures and program locations cannot be matched when tracing an error, we use novel techniques to identify program points where the data structures being traced should contain equivalent values. This technique also allows variables that have been renamed to be identified. The technique that allows Relative Debugging to be automated is presented in Chapter 5.
4 Data Flow Browsing for Relative Debugging

“If you don’t see the bug where you’re looking, then you’re looking in the wrong place.” - Loeser and Gaposchkin, 1976 [91].

Relative Debugging, as discussed in chapter 3, is a technique that assists developers to locate errors by comparing the contents of key data structures in a development version with the corresponding data structures in an existing, believed to be correct version. The technique requires users to manually identify the corresponding data structures and program points at which comparison should be made when the two programs are executed. In order to efficiently identify data structures and comparison points that will uncover the source of an error requires that users have a detailed knowledge of the two programs.

This chapter presents data flow browsing for Relative Debugging as an extremely useful technique that allows users to quickly and effectively define assertions that trace an error back to the origin of the fault, and reduces the need for users to have a detailed knowledge of the two programs. The first three sections of this chapter provide a detailed description, with a focus on the unique features, of a Relative Debugging Data Flow Browser. The following two sections present the design and implementation of a Relative Debugging Data Flow Browser. The chapter concludes with a case study that highlights that a Relative Debugging Data Flow Browser is an extremely useful tool when using Relative Debugging to identify errors in programs that have undergone software evolution.

4.1 What is a Data Flow Browser?

Data flow analysis is a process that is used to collect information about the data that a program may manipulate [21]. A range of techniques that collect information about a program’s data exist, varying in complexity and the types of information that is gathered. Data flow analysis is used for a number of applications, the most common being compiler optimization. Other examples include program comprehension, reverse engineering and refactoring [51].
A data flow browser presents the information gathered by the data flow analyses performed on a program. The information presented by the data flow analysis will depend on the information collected by the data flow analyses. The manner in which the information is presented will depend on the purpose of the application. For example, if a data flow browser was being used for program comprehension, the information needs to be presented in a clear and concise manner such that the user can obtain an understanding of the program.

4.2 Why Another Data Flow Browser?

The Relative Debugging methodology, presented in Section 3.1.3, defines the steps that a user should follow in order to localize faults. In general, if the output of a computation is reported to be incorrect, the user should define additional assertions to follow the inputs of the faulty computation. To do this the user will typically locate the definition points of the variables used in the computation. This process usually continues, in an iterative fashion, until the faulty section of code is located and traditional debugging techniques can be used to correct the error.

To locate variable definitions and construct useful assertions requires that the user has a detailed knowledge of the two programs under consideration. Without this, the task of locating definitions becomes difficult and time consuming. This is especially true in complex multi-file programs where there may be multiple definitions that are sparsely scattered throughout the program.

By applying the Relative Debugging methodology, it can be seen that the data flow properties of a program are needed to identify useful data structures and comparison points. Using the data flow properties of programs, we propose a Relative Debugging Data Flow Browser, which locates and displays the definitions that may assign a value to a selected variable in a particular computation. Such a tool provides invaluable support for Relative Debugging by reducing the complexity of identifying variable definitions. Furthermore, it decreases the need for users to have a detailed knowledge of the program under consideration, making Relative Debugging significantly easier than in the past, and subsequently improving productivity.
4.2.1 Example

For example, consider the two trivial program fragments shown in Figure 4-1. Suppose that an assertion comparing data structure \( Z \) on line 7 of the two programs indicates that there is a difference. In order to find the source of the error, it is necessary to place two further assertions at lines 2 and 5, each of which compares the two inputs, \( A \) and \( B \), to the expression. The results of these assertions will indicate which of \( A \) or \( B \), or both, requires further investigation. Currently, the information about the data-flow must be determined manually, that is, the programmer needs to inspect the expression on line 7 and locate all definitions that lead to this computation. Whilst these code fragments are trivial, and the flow can be seen easily, this is not always the case, especially in more complex programs.

<table>
<thead>
<tr>
<th>Reference Program</th>
<th>Suspect program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( A = f(...) )</td>
<td>( B = g(...) )</td>
</tr>
<tr>
<td>2: ( B = g(...) )</td>
<td>( A = f(...) )</td>
</tr>
<tr>
<td>3: ( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>4: ( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>5: ( A = h(...) )</td>
<td>( A = h(...) )</td>
</tr>
<tr>
<td>6: ( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>7: ( Z = A + B )</td>
<td>( Z = A + B )</td>
</tr>
</tbody>
</table>

Figure 4-1: Using data flow information to debug a program.

These code fragments highlight a further complication in building assertions. In order to determine whether the inputs of an expression are incorrect, such as \( A \) and/or \( B \), it is necessary to find their define points in the code. In the example above, the define points can easily be located because they are located a few lines from the computation. However, in complex multi-file programs, variable definitions may be sparsely scattered throughout the two programs which can make locating them complex and error prone. Further, in this example, \( A \) and \( B \) are defined in a different order in the two programs. While this may not affect the results of the computation, it is necessary to examine the code carefully in order to correctly generate assertions.
4.3 What’s Different about a Relative Debugging Data Flow Browser?

While data flow browsers are not in mainstream use, a number of implementations exist [20, 23]. To be versatile a Relative Debugging Data Flow Browser should extend the functionality currently found in data flow browsers with the following additional and unique features:

- Efficiency.
- Multi-lingual.
- Usability.

4.3.1 Efficiency

Any interactive debugging tool needs to be efficient and respond to user queries in a timely fashion. A Relative Debugging Data Flow Browser is no different, especially because a user will typically trace a number of potentially erroneous variables in order to create assertions during each refinement step.

Hence, an imperative design goal of any interactive debugging tool is the efficient processing of user queries and functions. Global program analysis techniques must therefore be avoided to ensure that the tool remains useable, scalable, and efficient. This is important because the tool is used interactively, and the user does not want to wait for the analysis to be performed across the whole program.

The majority of debugging and program analysis tools, such as those listed in Figure 4-2, perform the required program analysis across the entire program. This approach is time consuming and not suitable for a Relative Debugging Data Flow Browser. A Relative Debugging Data Flow browser should only generate, as requested, the information required to identify the variable definitions requested by the user.
### Purpose
- Debugging and program analysis

<table>
<thead>
<tr>
<th>Language</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Aristotle [62, 86]</td>
</tr>
<tr>
<td></td>
<td>CANTO [24]</td>
</tr>
<tr>
<td></td>
<td>ChopShop [72, 73]</td>
</tr>
<tr>
<td></td>
<td>CodeSurfer [23]</td>
</tr>
<tr>
<td></td>
<td>Ghinsu [90]</td>
</tr>
<tr>
<td></td>
<td>GUSTT [61]</td>
</tr>
<tr>
<td></td>
<td>Spyder [20]</td>
</tr>
<tr>
<td></td>
<td>Surgeon’s Assistant [52]</td>
</tr>
<tr>
<td></td>
<td>Unravel [92]</td>
</tr>
<tr>
<td>Cobol</td>
<td>IBM VisualAge for Cobol [69]</td>
</tr>
<tr>
<td>Java</td>
<td>Indus [74]</td>
</tr>
<tr>
<td>Oberon</td>
<td>Linz Oberon Slicing System [127]</td>
</tr>
<tr>
<td>Pascal</td>
<td>Osaka [104],</td>
</tr>
<tr>
<td></td>
<td>PELAS [80]</td>
</tr>
</tbody>
</table>

**Figure 4-2: Slicing tools.**

### 4.3.2 Multilingual

An important application of Relative Debugging is the debugging of programs that have been ported from one language to another. To ensure that a Relative Debugging Data Flow Browser remains consistent with this use, it needs to be capable of processing many languages.

Most debugging and program comprehension tools, such as those listed in Figure 4-2, are designed to be used with one particular programming language [20, 41, 120]. Significant redevelopment is therefore required if the tool is to be used with another language. In particular, the redevelopment will involve either modifications to the compiler or rewriting parsing routines. In either case, the result is undesirable development and maintenance costs. A Relative Debugging Data Flow Browser should preferably avoid a tight coupling with any one particular language.
4.3.3 Usability

Debugging software is a slow and error prone business. A programmer must locate errors by tracing the computations, usually at run time, in an attempt to observe the first point at which the program generates an erroneous or unexpected result. A user will typically use debugging software to control aspects of the program’s execution, including the ability to set break points and examine/modify program state. In addition to controlling the execution, the programmer must trace the flow of an error back through a program to identify the origin of the problem. Tracing the dataflow of a computation is both complex and error prone because the definitions of suspect variables may be sparsely scattered throughout the program. Further, issues such as side effects and aliasing of variables through function calls further complicate matters. Standard debuggers concentrate on run time issues, such as process control, and provide little support for tracing the static data flow properties of a program.

A Relative Debugging Data Flow Browser provides support for tracing the static data flow properties of a program. To ensure that a Relative Debugging Data Flow Browser remains an effective tool that allows users to quickly and efficiently trace the data flow of a program, it must:

- Be embedded in an IDE such that users can quickly navigate variable definitions in complex programs.
- Allow users to trace variable definitions whilst executing the program under debugger control.

4.4 Design of a Relative Debugging Data Flow Browser

The design of a Relative Debugging Data Flow Browser must address the requirements discussed in the previous section, namely efficiency, multi-linguality and usability. We have designed a Relative Debugging Data Flow Browser that meets these requirements by using a number of well known program analysis techniques.

4.4.1 Efficiency

A Relative Debugging Data Flow Browser needs to perform fast efficient program analysis when presenting variable definitions.
4.4.1.1 Use Define Chains

An efficient way to represent the definitions of selected variables is through *Use Define (UD) chains* [21, 29, 63]. UD chains provide a convenient structure to represent the set of definitions that may define the value of a variable used at a particular program point.

UD chains are commonly computed using the *Reaching Definitions* data flow analysis technique [21]. The Reaching Definition data flow analysis technique statically determines the set of variable definitions that may reach a variable at a given program point.

Figure 4-3 for example, shows the definitions that comprise the UD Chain for the variable \( x \) used in the statement on line 12. The UD chain for variable \( x \), used in the statement on line 12, is \{line 2, line 6, line 8\}.

```plaintext
1:  x = 5
2:  x = x * x
3:  if (p < 10)
4:      if (p < 5)
5:          x = 2
6:      x = x * 2
7:  else
8:      x = 3
9:  end-if
10: end-if
11: print x
```

Figure 4-3: Example UD Chain.
4.4.1.2 Demand Driven Analysis

The majority of debugging and program analysis tools, such as those listed in Figure 4-2, generate the exhaustive set of UD chains by using global analysis techniques [21]. A Relative Debugging Data Flow Browser should avoid the inherent expense of global analysis by adopting a demand driven approach [45, 68]. In particular, a Relative Debugging Data flow browser should only generate, as requested, the UD chains for those variables that are of interest to the programmer. A demand driven approach allows the Data flow browser to be efficient because time consuming, and possibly redundant, program analysis on the entire program is avoided. In addition, the generated information should be cached so that it may be used during the constructions of subsequent UD chains.

4.4.1.2.1 Related Work

Slicing is a technique that has been suggested as an aid for a number of software engineering tasks, such as debugging and program analysis, program differencing and integration, software maintenance and testing. The original notion of a program slice, however, was suggested for debugging purposes by Weiser in 1979 [135]. Weiser believed that a program slice corresponds to a mental abstraction that people make when they are debugging a program.

The set of program elements that may affect the value of a given variable, at a particular program point, is referred to a program slice [130]. The requested variable and associated program point is referred to as the slicing criterion and is typically represented by a pair \((p, v)\), where \(p\) is a program point and \(v\) is the variable. Hence, the program slice with respect to slicing criterion \(C\), is a subset of the statements and predicates that may directly, or indirectly, affect the values computed for the variable \(v\) at program point \(p\).

A static program slice is a program slice that has been computed without considering the program’s input. That is, the static program slice will contain every element that could be executed if the program was executed with every possible test case. A dynamic program slice, on the other hand, contains only the program elements that would be executed given a specific test case. For example, the statements shown in
bold font in Figure 4-4 represent the statements that comprise the static slice with respect to variable \( x \) used on the last line of the program. In contrast, the statements shown in bold font in Figure 4-5 represent the statements that comprise the dynamic slice for the same slicing criterion, if variable \( \text{pred} \) equals true.

| \( \text{a} = 10 \) | \( \text{a} = 10 \) |
| \( \text{b} = 6 \) | \( \text{b} = 6 \) |
| \( \text{if pred} \) | \( \text{if pred} \) |
| \( y = a + a \) | \( y = a + a \) |
| \( x = a + b \) | \( x = a + b \) |
| \( b = b - 1 \) | \( b = b - 1 \) |
| \( \text{else} \) | \( \text{else} \) |
| \( y = b + b \) | \( y = b + b \) |
| \( b = b + 1 \) | \( b = b + 1 \) |
| \( x = a * b \) | \( x = a * b \) |
| \( b = b + 1 \) | \( b = b + 1 \) |
| \( \text{end-if} \) | \( \text{end-if} \) |
| \( y = y / 2 \) | \( y = y / 2 \) |
| \( \text{print b} \) | \( \text{print b} \) |
| \( \text{print x} \) | \( \text{print x} \) |

Figure 4-4: Static program slice with respect to variable \( x \) on the last line.

Figure 4-5: Dynamic program slice with respect to variable \( x \) on the last line and \( \text{pred} = \text{true} \).

A Relative Debugging Data Flow Browser does not need to highlight program slices because the user is only interested in variables that are incorrect. Instead, a Relative Debugging Data Flow Browser should display the set of definitions that may define the value of a specific variable at a particular program point.

Spyder [20] and CodeSurfer [23] are interactive debugging tools that highlight program slices for a specified variable at a given program point. These tools also locate and highlight the set of definitions that reach a given variable at a particular program point. However, as shown in Figure 4-2, Spyder and CodeSurfer are limited to programs written in one language, i.e., C language, and require modification to an exist-
ing compiler. Both of these requirements, or limitations, are not desirable for Relative Debugging purposes.

4.4.2 Multilingual

To avoid the tight coupling with the high level programming language, we propose that a Relative Debugging Data Flow Browser performs the required program analysis on an intermediate language. Virtual machines, such as Sun’s Java Virtual Machine (JVM) [89] and Microsoft’s NET Common Language Runtime (CLR) [101], that execute portable binaries containing generic intermediate code have been widely accepted. Performing program comprehension directly on the intermediate code avoids the tight coupling with the high level programming language. The tool is not bound to one particular programming language, but rather functions with any programming language that targets the intermediate language. Hence, the program comprehension tool can be used with more than one language without requiring any modifications.

For example, Java, Ada95 and Component Pascal may all be compiled to Java bytecode [53, 109]. A tool that performs program analysis on the emitted bytecode does not need to consider the different syntax for each of these languages. The same situation exists for C++, C# and VB, among others, that can be compiled to Microsoft’s intermediate language (MSIL) [102].

4.4.2.1 Related Work

Recent investigation regarding program analysis of intermediate languages has focused on traditional global analysis [28, 93, 128, 145]. However, performing interactive analysis on programs translated to intermediate languages has received little consideration. The work in [82] details the construction of control flow graphs, symbol table information, and reaching definitions data from Java bytecode but the technique is not interprocedural or demand driven.

4.4.3 Usability

To remain usable, a Relative Debugging Data Flow Browser should be embedded into an IDE to deliver the following features:

- Allow a user to hover over a variable, and request the definitions that reach it.
• Should clearly list the definitions that reach the selected variable.
• Allow the user to quickly and easily navigate to each of the located definitions.
• Should be able to be used while under the control of a Relative Debugger.

4.4.4 Algorithm

As discussed above, our Relative Debugging Data Flow Browser uses UD chains to represent and present the set of definitions that may reach a variable at a particular program point. And, to remain multi-lingual, and language independent, our Relative Debugging Data Flow Browser performs the analysis directly on the intermediate language.

To identify variable definitions, the Relative Debugging Data Flow Browser needs to analyze the program containing the intermediate language and identify each instruction that could assign a value to the variable of interest at a particular program point. When performing such analysis, the Relative Debugging Data Flow Browser cannot rely on the high level language grammar nor can it use information or data structures available at compile time, such as abstract syntax trees, symbol tables or control flow graphs, that are normally built and maintained by the compiler. The Relative Debugger Data Flow Browser must therefore build the data structures required to construct UD chains by processing the intermediate instructions emitted by the compiler. The data structures required to construct UD chains are summarized in Figure 4-6.

To locate the definitions that reach the selected variable, the CFG must be traversed back from the basic block that contains the selected variable. If this block contains a definition for the selected variable, prior to the use, it is the only definition that may reach the selected use (due to the properties of a basic block [21]). If the start block does not contain a definition for the selected variable it is possible that more than one definition may reach the use, along different control paths. Therefore, each control path that leads to the starting block needs to be scanned backwards for the last definition. This is achieved by recursively scanning the predecessors of the current block.
<table>
<thead>
<tr>
<th><strong>Data Structure</strong></th>
<th><strong>Purpose</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow graph</td>
<td>A control flow graph (CFG) provides the information that makes it possible to locate the last definition from each control path that leads to the selected variable. The CFG can be constructed using conventional techniques [21].</td>
</tr>
<tr>
<td>Call graph</td>
<td>A call graph [60] needs to be constructed to allow UD chains to be constructed across function boundaries. A call graph represents the possible transfer of control between functions.</td>
</tr>
<tr>
<td>Class hierarchy</td>
<td>A class hierarchy [25, 40] is constructed to allow all possible definitions to be located when a virtual call is encountered. That is, during UD chain construction each overriding function in each subtype must be scanned for possible variable definitions.</td>
</tr>
</tbody>
</table>

**Figure 4-6: Data structures required to construct UD chains.**

If a call instruction is encountered along a path during UD chain construction and the variable for which the UD chain is being constructed is passed by reference to the callee, then UD chain construction must continue in the called method. In this case, the data structures detailed in Figure 4-6, for the called method, are initialized and UD chain construction continues from the last block in the constructed CFG. The control paths in the called method are searched (backwards) for definitions of the corresponding formal parameter.

If each control path in the called method contains a definition for the corresponding formal parameter, then UD chain construction along the current path at the callsite can cease. However, if not every control path through the called method provides a definition for the corresponding formal parameter, UD chain construction must continue along the current control path at the callsite (from the instruction prior to the call).
In addition, if the call is to a virtual function, then the overriding functions in each subclass need to be processed in the same manner. The constructed class hierarchy is used to determine the overriding methods that are processed in each subclass.

If the beginning of the method has been reached along a control path during UD chain construction of a parameter, then not all control paths within the method define a value for the incoming parameter. In this case, UD chain construction must continue at each callsite of the current method, as defined in the constructed call graph, of the current method.

4.4.4.1 Example

A simple example is illustrated with Figure 4-7 and Figure 4-8. The control flow graph (CFG) in Figure 4-8, representing the simple program listed in Figure 4-7, consists of five basic blocks; the flow of control between the blocks is represented by the solid lines. Assuming that the user requests the definitions that lead to variable $x$ on line 22, the Relative Debugging Data Flow Browser commences UD chain construction from block 04 in the CFG. The dotted lines in Figure 4-8 show the order in which the Relative Debugging Data Flow Browser traverses the CFG during UD chain construction. Figure 4-8 also shows the variable definitions that have been identified in each basic block.
```csharp
public void Example(int num, int div) {
    int x = 0;
    int y = 14;
    if (div > 0) {
        if (num/div == 2) {
            x = num;
            x = x / 2;
        }
    }
    else {
        x = 1;
    }
    Console.WriteLine(x);
}
```

Figure 4-7: Program for UD Chain construction example.

Figure 4-8: Control flow graph for program listed in Figure 4-7.
4.5 DUCT: A Data Flow Browser for Relative Debugging

We have developed a Relative Debugging Data Flow Browser, named DUCT\textsuperscript{1} \cite{DUCT}, that finds and highlights the definitions of a selected variable at a specified program location. The implementation is based on the design discussed in the previous section.

4.5.1 Architecture

DUCT generates UD chains for programs that execute within the Microsoft .NET framework and displays the results within the Microsoft Visual Studio environment \cite{VisualStudio}. A .NET program is contained within a portable executable (PE) file that contains compiler generated Microsoft Intermediate Language (MSIL) \cite{MSIL} and metadata \cite{Metadata} that conforms to the Common Type Specification (CTS) \cite{CTS}. MSIL is a CPU independent instruction set that can be efficiently transformed into native code (by a just-in-time (JIT) compiler) at runtime. Metadata allows a .NET program to be self-describing by detailing the types, signatures, and other data that the .NET framework uses.

In practice, this allows DUCT to be used with a larger number of development languages (i.e., any language that targets the Microsoft.NET framework). For research purposes, the framework provides a common, well defined platform in which the tool and exercised techniques can be evaluated. To date, DUCT has been successfully tested on .NET programs built by the GPCP \cite{GPCP} compiler as well as the Microsoft compilers for VB, C#, C++, J#.

DUCT has been integrated into Microsoft Visual Studio .NET \cite{VisualStudio} using the Visual Studio Integration Program (VSIP) \cite{VSIP}. VSIP is a component framework that allows Visual Studio .NET to be extended. The framework allows DUCT to be hosted by Microsoft Visual Studio .NET and utilize the IDE and services offered by the environment.

\textsuperscript{1} The name DUCT, an acronym for Define Use Chain Tool, is a misnomer. The tool actually locates use define (UD) chains, not define use (DU) chains. We elected to keep the name as no better alternative could be found before the tool was released.
VSIP allows Microsoft Visual Studio .NET to be extended by providing custom functionality in VSPackages. VSPackages are components that implement interfaces defined by VSIP, allowing Microsoft Visual Studio .NET and VSPackages to interact. VSPackages can proffer services that other VSPackages consume. Microsoft Visual Studio .NET also exposes a number of VSPackages that proffer a broad range of services and provide access to the environment's windowing functionality. Figure 4-9 illustrates the integration of DUCT and VSGuard within the VSIP framework.

Figure 4-9: Extending Microsoft Visual Studio .NET with DUCT and VSGuard.
4.5.2 User Interface

The DUCT and VSGuard packages contain the implementation details that extend Microsoft Visual Studio .NET with new functionality. These packages also describe how Microsoft Visual Studio .NET’s user interface is extended so that the user can access the new functionality.

4.5.2.1 DUCT

To locate the definition(s) reaching a variable, the user must select the variable in the source program and inform DUCT to locate the definitions. This is achieved by:

- Placing the cursor over the variable of interest.
- Selecting ‘Construct UD Chain’ on the pop up context menu, which is activated by clicking the right mouse button.

DUCT currently recognizes scalar variables, objects, object member fields, and arrays. The located definition(s) are clearly highlighted in the source code window of the IDE and listed in a separate window, allowing the user to easily navigate through the definitions.

For example, Figure 4-10 illustrates the UD chain constructed by DUCT for the variable $x$ on line 22 in a simple C# program. Each definition that reaches the use of $x$ on line 22 is highlighted in the source code window and detailed in the output window at the bottom of the IDE. The user may navigate to a particular definition by clicking on the definition in the output window. This is particularly useful when a definition is not displayed on the screen or is located in another file.
Figure 4-10: Example DUCT screenshot.
4.5.2.2 VSGuard

Once DUCT has identified the definitions that reach the selected variable, the user will typically create one or more VSGuard assertions. The user creates a VSGuard assertion by:

- Right clicking on a variable in either the original or evolved program.
- Selecting ‘Add To Assert’ from the pop up context menu, which is activated by clicking the right mouse button. A half filled Edit Assert form will be displayed.
- Placing the cursor on the corresponding variable in the second program and select ‘Add To Assert’ from the pop up context menu. The other half of the ‘Edit Assert’ window will be populated with the corresponding information from the second program. The ‘Edit Assert’ window is now filled with information describing the assertion. Figure 4-11 shows an example of the Edit Assert window.
Once the user is satisfied with the assertions, VSGuard can be started to identify which assertions are failing and reveal the source of an error or require further attention. The results of each assertion are displayed in individual Assertion Results windows, similar to the windows shown in Figure 4-12.

![Figure 4-12: VSGuard Assertion Results Window.](image)

### 4.5.3 Implementation Details

DUCT adopts a demand driven approach for the construction of UD chains. The UD chain is constructed when requested, and only computes the minimal information required to facilitate the construction. In addition, the generated information is cached so that it avoids re-generation if it should again be required during the construction of subsequent UD chains.

As discussed in Section 4.4.2, a Relative Debugging Data Flow Browser should perform the required program analysis on an intermediate language to remain language neutral. DUCT has been developed to process the Microsoft Intermediate Language (MSIL) [102]. MSIL is a CPU and platform independent instruction set that is exe-
cuted by a virtual, stack based machine. Compilers that target the .NET framework translate the source code into MSIL.

A MSIL instruction consists of an opcode and zero or more operands. The opcode is the portion of the instruction that specifies the operation to be performed; the operand specifies the data on which the operation should be performed. For example, the first 2 bytes, FE0E, of the MSIL instruction shown in Figure 4-13 represent the opcode and, the last 2 bytes, 0011, represent the operand. In this case, the opcode FE0E represents the stloc operation and specifies that the value on the top of the stack be popped and stored in 17th local variable.

![Figure 4-13: Microsoft Intermediate Language (MSIL) example.](image)

To build the required data structures, as listed in Figure 4-6, DUCT utilizes the metadata and symbolic debug information [106] when constructing UD chains. The metadata is used to obtain information about types that the code defines and the types that it references externally. In order to present the constructed UD chains in a usable manner that the user can quickly understand, DUCT relies on the symbolic debug information, stored in the symbol store. The symbol store contains the information that allows DUCT to interact with the user in relation to the high level source program.

### 4.5.3.1 Generating Intraprocedural UD Chains

To construct the UD chain for a selected variable, in a demand driven manner, DUCT performs the following steps:

1. **Step 1.** Determine the method in which the user has placed the cursor.
2. **Step 2.** Extract the information required to construct the requested UD chain.
a. Extract the method’s local variables and parameters.

b. Extract the token bound by the cursor.

c. Extract the source line number associated with each MSIL instruction.

Step 3. Construct the data structures required to construct the requested UD chain.

a. Construct the method’s control flow graph (CFG).

Step 4. Traverse and scan the CFG for the last variable definition from each control path that leads to the selected variable.

Step 1: To determine the variable selected by the user, DUCT must determine the method in which the cursor resides. Each method in a .NET program is uniquely identified by a metadata token. To retrieve the metadata token associated with the current method, DUCT queries the source code window using interfaces exposed by VSIP. The metadata token allows DUCT to subsequently retrieve information required from the metadata and symbol store.

Step 2a: The metadata token, obtained in step 1, can be used to extract the method’s local variables and parameters. The details of the local variables and parameters are required for two reasons. First, the information is used to determine if the user has actually selected a token that represents a variable within the method’s local scope. Second, DUCT needs to map the high level variable name to the compiler assigned index number that identifies the variable in the program’s MSIL. The index number is used by DUCT to identify store instructions that update the memory allocated for the selected variable (discussed in Step 4). The local variable and parameter information are retrieved from the symbol store and metadata respectively.

Step 2b: The token representing the selected variable is extracted from the source code window using interfaces exposed by VSIP. Once the token has been extracted, DUCT ensures that the selected token actually represents a variable within the current method’s local scope. This is achieved by ensuring that the selected token exists within the list of local variables and parameters extracted in the previous step.
**Step 2c:** Each source line in the high level program is associated with one or more MSIL instructions in the compiled .NET program. Determining the source line associated with each MSIL instruction allows DUCT to report variable definitions in terms of the source line in the high level program. Such line numbers cannot readily be retrieved on demand. Therefore, the line number to which each instruction belongs must be extracted, from the symbol store, for the entire method prior to UD chain construction.

**Step 3a:** The method’s control flow graph (CFG) [21] must be constructed before UD chain construction can commence. The CFG provides the information that makes it possible to locate the last definition from each control path that leads to the selected variable.

The CFG is constructed using conventional techniques [21]. The first step in building a CFG is determining the basic blocks in the procedure. A basic block is a sequence of instructions where the flow of control can only be entered at the beginning of the block, and exit at the end of the block. The CFG is then constructed by connecting the basic blocks to represent the flow of control between blocks. Figure 4-14 lists the MSIL instructions that define the boundaries of a basic block in a .NET method and must be identified when constructing the basic blocks.
### Instruction Description

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>Branch on equal.</td>
</tr>
<tr>
<td>bge</td>
<td>Branch on greater than or equal to.</td>
</tr>
<tr>
<td>bgt.un</td>
<td>Branch on greater than or equal to, unsigned or unordered.</td>
</tr>
<tr>
<td>bgt</td>
<td>Branch on greater than.</td>
</tr>
<tr>
<td>bgt.un</td>
<td>Branch on greater than, unsigned or unordered.</td>
</tr>
<tr>
<td>ble</td>
<td>Branch on less than or equal to.</td>
</tr>
<tr>
<td>ble.un</td>
<td>Branch on less than or equal to, unsigned or unordered.</td>
</tr>
<tr>
<td>bne.un</td>
<td>Branch on not equal or unordered.</td>
</tr>
<tr>
<td>br</td>
<td>Unconditional branch.</td>
</tr>
<tr>
<td>brfalse</td>
<td>Branch on false, null, or zero.</td>
</tr>
<tr>
<td>brtrue</td>
<td>Branch on non-false or non-null.</td>
</tr>
<tr>
<td>leave</td>
<td>Exit a protected region of code.</td>
</tr>
</tbody>
</table>

**Figure 4-14: MSIL control transfer instructions.**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>starg</td>
<td>Stores a value to the argument numbered <em>num</em>. The value is retrieved from the stack.</td>
</tr>
<tr>
<td>stelem</td>
<td>Stores a value in an element of an array. The array, element index, and value are retrieved from the stack.</td>
</tr>
<tr>
<td>stind</td>
<td>Stores a value of type <code>&lt;type&gt;</code> into memory. The memory address and value are retrieved from the stack.</td>
</tr>
<tr>
<td>stfld</td>
<td>Stores a value into the field <em>field</em> of an object. The object and value are retrieved from the stack.</td>
</tr>
<tr>
<td>stloc</td>
<td>Stores a value into local variable <em>index</em>. The value is retrieved from the stack.</td>
</tr>
</tbody>
</table>

**Figure 4-15: MSIL store instructions.**
**Step 4**: Once the CFG has been constructed, DUCT constructs the UD chain by locating the definitions that reach the selected variable. DUCT locates the variable definitions by identifying MSIL store instructions that update the memory allocated for the selected variable. The MSIL store instructions that DUCT must identify are listed in Figure 4-15.

To determine if the current block contains a definition for the selected variable, DUCT identifies all MSIL store instructions in the current block. The target variable for each store instruction must be determined. All store instructions, except for indirect stores, implicitly encode the target variable and hence, can be extracted directly from the instruction. In contrast, indirect store instructions receive the target address of the variable from the stack. Therefore, to correctly locate the load instruction that pushes the address of the target variable onto the stack, DUCT must perform symbolic execution of the abstract stack machine. For example, Figure 4-16 illustrates a store instruction, `stind.i4`, that needs to determine the target variable by performing symbolic execution of the abstract stack machine. In particular, the target variable is represented by the `ldarg.1` instruction, which pushes the address of the method’s first parameter.

```
// x = x * 2;

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldarg.1</td>
<td>pushes the memory address used by the stind instruction - in this case, the method’s first parameter</td>
</tr>
<tr>
<td>ldarg.1</td>
<td>pushes the value used by the following mul instruction - in this case, pushes x and 2onto the stack</td>
</tr>
<tr>
<td>ldind.i4</td>
<td>onto the stack</td>
</tr>
<tr>
<td>ldc.i4.2</td>
<td>onto the stack</td>
</tr>
<tr>
<td>mul</td>
<td>pushes the result of the mul instruction onto the stack</td>
</tr>
<tr>
<td>stind.i4</td>
<td>stores the value that was pushed onto the stack into the variable’s address that has was also pushed onto the stack i.e., the result of mul instruction is placed in the address of the method’s first parameter</td>
</tr>
</tbody>
</table>
```

**Figure 4-16: Indirect store instruction requiring symbolic execution of the abstract stack machine.**
To locate the definitions that reach the selected variable the CFG must be traversed back from the basic block that contains the selected variable. If this block contains a definition for the selected variable, prior to the use, it is the only definition that may reach the selected use (due to the properties of a basic block [21]). If the start block does not contain a definition for the selected variable it is possible that more than one definition may reach the use, along different control paths. Therefore, each control path that leads to the starting block needs to be scanned (backwards) for the last definition. This is achieved by recursively scanning the predecessors of the current block. When a definition for the selected variable is found, the definition is added to the UD chain and searching along the current control path can cease.
4.5.3.1.1 Example

The following example illustrates the steps performed by DUCT, as discussed in the previous section, when constructing an intraprocedural UD chain for a selected local variable.

If a user would like to locate the definitions that may define a value for the local variable ‘x’ used on line 22, in the C# method shown in Figure 4-17, DUCT would proceed by extracting the method details, as outlined in Steps 1 and 2 above. The extracted details, as described in Step 2, are shown in Figure 4-18.

```csharp
07:   public void Example(int num, int div)
08:   {
09:      int x = 0;
10:      int y = 14;
11:
12:      if (div > 0) {
13:         if (num / div == 2) {
14:            x = num;
15:            x = x / 2;
16:         }
17:      }
18:      else {
19:         x = 1;
20:      }
21:
22:      Console.WriteLine(x);
23:   }
```

Figure 4-17: Example C# method.
**MethodInfo for Example**

Method name: Example
MD Token: 0x06000001
Class: CSEExample.Class1
Virtual: False

---

**Parameters for Example**

Parameter 00: this [ByRef]
Parameter 01: num
Parameter 02: div

---

**Var/Index for Example**

div 0x08000002 int32
num 0x08000001 int32
this 0x08000000 Unknown Type
x 0x00000000 int32
y 0x00000000 int32

---

**Opcode/Line Map for Example**

| 00: L0009 | 13: L0013 | 26: L0015 |
| 01: L0009 | 14: L0013 | 27: L0019 |
| 02: L0010 | 15: L0013 | 28: L0019 |
| 03: L0010 | 16: L0013 | 29: L0022 |
| 04: L0010 | 17: L0013 | 30: L0022 |
| 05: L0012 | 18: L0014 | 31: L0022 |
| 06: L0012 | 19: L0014 | 32: L0022 |
| 07: L0012 | 20: L0014 | 33: L0022 |
| 08: L0012 | 21: L0015 | 34: L0022 |
| 09: L0012 | 22: L0015 | 35: L0023 |
| 10: L0013 | 23: L0015 |
| 11: L0013 | 24: L0015 |
| 12: L0013 | 25: L0015 |

---

Figure 4-18: Extracted method details for method listed in Figure 4-17.
The constructed CFG, as discussed in Step 3a, is shown in Figure 4-19. The solid lines represent the flow of control between the basic blocks. DUCT commences UD chain construction from block 5 in the CFG. The broken lines in Figure 6 shows the order in which DUCT traverses the CFG during UD chain construction. The figure also shows the store instructions that have been extracted from each block and highlights, in bold, the instructions that represent definitions of the selected variable \( x \).

![Figure 4-19: Control flow graph and extracted method details for program listed in Figure 4-17.](image-url)
Figure 4-20 shows the results produced by DUCT within Microsoft Visual Studio.NET. Figure 4-21 highlights DUCT’s ability to operate with multiple languages by showing the UD chain produced by DUCT for a similar program written in Visual Basic .NET.

Figure 4-20: Definitions located by DUCT in the program listed in Figure 4-17 (screenshot).
Figure 4-21: Definitions located by DUCT in a VB .NET program (screenshot).
4.5.3.2 Generating Interprocedural UD Chains

The intraprocedural design can be easily extended to provide a framework for constructing interprocedural UD chains. In addition to the steps performed for intraprocedural UD chains, the following steps are required to construct interprocedural UD chains:

**Step 3.** Construct the information required to construct the requested UD chain:

a. Construct the method’s CFG.

b. Construct the program’s call graph.

c. Construct the program’s class hierarchy.

**Step 4.** Traverse and scan the CFG, and if required the call graph, for the last variable definition from each control path that leads to the selected variable.

**Step 3a:** The CFG may be constructed in a similar manner to that required for intraprocedural UD chains. For the construction of interprocedural UD chains however, a call instruction must also define the boundary of a basic block. This additional requirement allows the flow of control between methods to be considered during UD chain construction. Figure 4-22 lists the MSIL call instructions that define the boundaries of a basic block in a .NET method.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>Call a method.</td>
</tr>
<tr>
<td>calli</td>
<td>Indirect method call.</td>
</tr>
<tr>
<td>callvirt</td>
<td>Call a method associated, at runtime, with an object.</td>
</tr>
</tbody>
</table>

*Figure 4-22: MSIL call instructions.*
DUCT builds a call graph [60] to allow UD chains to be constructed across method boundaries. A call graph represents the possible transfer of control between methods. A node in the call graph represents a method and an edge (p → q) exists if method p invokes method q.

A common use of call graphs is data flow analysis [60]. To assist with forward data flow analysis the edges in the call graph are created from the callsite to the callee (p → q). As UD chain construction is a backward data flow problem, DUCT actually creates the edges from the callee to the callsite (q → p).

**Step 3b:** The information provided by the constructed call graph is insufficient to locate all possible definitions in the presence of virtual calls. When a virtual call is encountered the possible receiver may be any of the declared subtypes. During UD chain construction each overriding method in each subtype must be scanned for possible variable definitions. A class hierarchy [25, 40] is constructed to allow DUCT to locate and traverse each overriding method.

**Step 3c:** The class hierarchy can be efficiently constructed using the program’s metadata. DUCT traverses each class defined in the program and records the inherited base class(es). While the demand driven approach provides quick response times an initial, one-off, analysis of the entire program is required to construct the call graph. Specifically, an initial sweep of the program is required to locate the call instructions in order to build the call graph. The construction of the call graph is therefore linear to the size of the program. On the other hand, the class hierarchy can be quickly constructed from the information contained within the metadata and the CFG is only constructed when a method is scanned for potential definitions.

**Step 4:** An interprocedural UD chain is constructed in a similar manner to that described for an intraprocedural UD chain. However, additional consideration is needed when the following program points are encountered:

- Callsites.
- Method entry.
The first consideration occurs when a call instruction is encountered (along a path) during UD chain construction. If the variable for which the UD chain is being constructed is passed by reference to the callee, then UD chain construction must continue at the called method. In this case, details of the called method are extracted (Steps 2 and 3) and UD chain construction continues from the last block in the constructed CFG of the called method. The control paths in the called method are searched (backwards) for definitions of the corresponding formal parameter.

If each control path in the called method contains a definition for the corresponding formal parameter, then UD chain construction along the current path at the callsite can cease. However, if not every control path through the called method provides a definition for the corresponding formal parameter, UD chain construction must continue along the current control path at the callsite (from the instruction prior to the call).

In addition, if the call is to a virtual function then overriding methods in each subclass need to be processed in the same manner. The class hierarchy constructed in Step 3c is used to determine the overriding methods in each subclass.

The second consideration occurs when the beginning of the method has been reached, along a control path, during UD chain construction of a parameter. This situation implies that the current control path within the method does not define a value for the incoming parameter. In this case, UD chain construction must continue at each callsite of the current method, determined from the call graph constructed in Step 3b.
4.6 Case Study

The benefits of using DUCT when performing Relative Debugging were highlighted in a real world case study. In this example, errors arose when the Earth program [3] was upgraded from Visual Basic version 6 to Visual Basic .NET. The Earth program is a free program that uses the VSOP87 planetary theory to compute the heliocentric ecliptic longitude, latitude, and distance to the sun of the planet Earth over a period of several thousand years.

After running the original and upgraded versions with identical inputs we noted, as illustrated in Figure 4-23 and Figure 4-24, that the resulting output was different. We proceeded to debug the error by locating the code that output the erroneous result (shown in Figure 4-25). This code indicated that the incorrect result was being produced by the value of variable \( Q \). The value of \( Q \) is assigned its value by the function call to \texttt{EARTH\_LBR\_FOR} on the previous line. We placed an assertion on the input parameter, \( Q \), to determine if the input parameter contained the same value in both versions.

![Figure 4-23: DUCT case study - result produced by the original Earth VB program.](image)

![Figure 4-24: DUCT case study - result produced by the ported Earth VB.NET program.](image)
Private Sub ComputeButton_Click(_
    ByVal eventSender As System.Object,
    ByVal eventArgs As System.EventArgs) _
Handles ComputeButton.Click

    Dim Q As Object
    JDE_FOR(INTERFACE_DATE, INTERFACE_TIME, Q)
    Q = EARTH_LBR_FOR(Q)
    Text2.Text = Q
End Sub

Figure 4-25: DUCT case study - code producing the erroneous result.

Next, we used DUCT to locate the definition(s) that assign a value to input parameter Q. Using the result produced by DUCT, illustrated in Figure 4-26, we placed assertions on the variables, W and Q, used in the right hand expression of the definition located in the JDE_FOR function, as shown in Figure 4-27.

Figure 4-26: DUCT case study – using DUCT to locate definitions for the input parameter Q.
Public Function JDE_FOR(
    ByRef Date_String As Object, _
    ByRef Time_String As Object, _
    ByRef fracRes As Object)
    ' Returns the fraction of a day corresponding to the given
    ' time argument in the standard "HH:MM:SS.sss" format.

    Dim Q, W As Object
    JD_NUM_FOR(Trim(Date_String), W)
    Q = Trim(Time_String)
    Q = (Val(Left(Q, 2)) * 3600.0# + Val(Mid(Q, 4, 2)) * _
         60.0# + Val(Mid(Q, 7, 16))) / 86400.0#
    fracRes = W + Q
End Function

Figure 4-27: DUCT case study – code containing definitions located by DUCT.

After creating these assertions we ran VSGuard to determine which variable contained an incorrect value. VSGuard identified, as shown in Figure 4-28, that the variable W in the function JDE_FOR contained different values in the two programs.

Figure 4-28: DUCT case study - VSGuard result showing that variable W differed.

Using DUCT again we located the definition(s) that assigns a value to W used in the statement fracRes = W + Q in the JDE_FOR function. The results are shown in Figure 4-29. Navigating to the definition located by DUCT, which resides in the JD_NUM_FOR function (shown in Figure 4-30), we continued the process and placed assertions on the define points of variables MM, MMM, Pointer, Q and DD. This allowed us to trace back through the expressions and determine that DD was correct but Len(DD) was incorrect. 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 the length of that string is returned. However, in Visual Basic .NET, the Len function takes an Ob-
ject as a parameter, and it returns the length of the object – in this case, a value of 8. The code was corrected by explicitly converting DD to a string as shown in Figure 4-31. This error occurred because of the subtle differences between two inbuilt library functions, and would have been much harder to locate without a tool such as DUCT, in combination with a Relative Debugger.

Figure 4-29: DUCT case study - using DUCT to locate definitions for the input parameter W.
Public Function JD_NUM_FOR(_
    ByRef DD_MMM_YYYY_BCAD As Object, _
    ByRef astroJDnum 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, "JANFEBMARAPRMAYJUNJULAUGSEPOTNOVDEC", 
        MMM) - 1) / 3))
    ...
    JD = DD + Int(367 * (MM + (Q * 12) - 2) / 12) + Int(1461 * (YYYY + 
        4800 - Q) / 4) - 32113
    ...
    astroJDnum = JD - 0.5
End Function

Figure 4-30: DUCT case study - JD_NUM_FOR function.

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

Figure 4-31: DUCT case study - required correction.
In this example, DUCT was instrumental in following the chain of errors back to the source. It allowed us to navigate the UD chains quickly, and thus helped us to determine where to place assertions. In all, it only took 4 iterations to locate the error, and a total of 11 assertions were required.

Although DUCT proves to be a useful debugging tool in its own right, our primary motivation for building such a tool was to evaluate the functionality and techniques used with an aim to automate Relative Debugging. For this reason, the tool is not complete and currently does not handle the following:

- DUCT does not perform alias analysis. Traditionally, such analysis is global and does not suit the demand driven approach.
- DUCT does not currently handle delegates (function pointers).
- DUCT cannot process multi-threaded programs. Difficulties arise in a multi-threaded program, because the definition(s) for a global variable may reside on different control paths than the one being considered.

### 4.7 Conclusion

This chapter has presented a Relative Debugging Data Flow Browser and demonstrated that such a data flow browser enhances the Relative Debugging paradigm. It provides invaluable support when using Relative Debugging to isolate errors in programs that have undergone software evolution; users can quickly identify and navigate variable definitions, allowing assertions that trace the flow of errors to be efficiently constructed.

Further, a Relative Debugging Data Flow Browser is not only a very useful tool, it also provides the ability to automate the process of Relative Debugging and further enhance the paradigm. As described in the following chapter, a Relative Debugging Data Flow Browser allows variable definitions to be identified and assertions to be constructed automatically.
5 Automatic Relative Debugging

"Debugging a new release of a system with the current working version while answering diagnosis queries may also result in significant labor." - Shapiro, 1982 [119].

As discussed in Chapter 3, Relative Debugging requires the user to manually identify the corresponding data structures and program points at which comparison should be made when two programs are executed. To identify where the programs first begin to diverge, and identify the origin of an error, the user will typically create assertions that trace the data flow leading back to the erroneous variable. The methodology for deciding where to place assertions is built around following the data and control flow of the two programs. For example, if the output of a computation is reported to be incorrect, the user typically defines additional assertions to follow the inputs to the faulty computation – typically by finding the definition points of the variables used in the computation. This process usually continues, in an iterative fashion, until the faulty section of code is localized and traditional debugging techniques can be used to correct the error.

This chapter presents an approach that automates the Relative Debugging methodology by identifying the data structures and program points for the user. In a manner very similar to the manual approach, we propose a systematic approach that constructs assertions to trace the flow of an error back to the origin of the fault. Our approach requires little human intervention and minimizes the need for users to have a detailed knowledge of the two programs under consideration. Such automation radically improves the current method for debugging programs that have undergone software evolution and dramatically improves productivity.

This chapter presents our research that has allowed the Relative Debugging methodology to be automated. A number of existing techniques have been leveraged to achieve automatic Relative Debugging and this has resulted in the following contributions:
• An approach that allows matching data structures and program locations to be identified such that assertions can be automatically constructed. The data flow properties of the two programs under consideration are used to automatically identify the data structures and program points where comparisons should be performed when the two programs are executed. In particular, our approach traces the data flow of two programs, identifies matching data structures and automatically constructs assertions. This chapter describes the techniques we have adopted to trace the data flow of two programs, and automatically construct assertions that will lead to the source of an error. Using the data flow of two programs in this manner raises a number of interesting issues that have been addressed by our research and are also presented in this chapter.

• An approach that allows suitable data structures and program locations to be identified when the programs differ and matching data structures cannot be identified when tracing an error back through a program. Using the data flow properties of two programs to automatically construct assertions and trace erroneous data structures is made easier when the data flow of the two programs is similar. When the data flow differs between the two programs and data structures cannot be matched, it is not possible to generate further assertions that trace the error. In this case, we propose a novel approach that allows matching data structures and program points to be identified earlier in the two programs, and provide a suitable point from which the automatic construction of assertions can resume.

• An approach that allows renamed variables to be matched such that assertions can be constructed automatically. Variables may be renamed as a program evolves. We present a novel approach that allows renamed variables to be identified so that assertions can be constructed in such cases.
Our research has resulted in the application of novel techniques that allows the Relative Debugging paradigm to be automated. The first section of this chapter presents the technique that allows the Relative Debugging to be automated. The second part of this chapter presents the design and implementation of an Automatic Relative Debugger as based on our research. The application of our Automatic Relative Debugger is then demonstrated, and shown to be extremely effective, in the following chapter.

### 5.1 Automatically Generating Assertions

In a manner very similar to the manual approach, we propose a systematic approach that constructs assertions to trace the flow of an error and isolate the origin of a fault. To effectively automate Relative Debugging, a combination of static and dynamic program analysis is required. Static analysis is used to identify suitable and corresponding data structures, and program points within two functions\(^2\). Dynamic analysis is used to determine which functions to proceed from when the source of the error does not reside inside the functions currently being considered.

Figure 5-1 illustrates the systematic approach that allows the Relative Debugging to be automated. The approach requires the user to identify the starting variables in each program that should contain the same values after the two programs have executed.

The approach commences by identifying the definitions that lead to the starting variables and automatically creating one representative assertion. The technique, and the reason that one representative assertion is created, rather than an assertion for each definition, is described in Section 5.1.1. The definitions that lead to the variables used in the definitions that led to the starting variables are then processed in a similar manner and assertions are automatically generated. This process continues until the functions that contain the two starting variables have been processed, and an assertion has been created for each variable that may influence the computation of the key variables under investigation.

---

\(^2\) Function is used to refer to a function, method, or subroutine.
The variables used in the definitions located in the reference program are matched with the variables in the definitions located in the development program based on name or behaviour. If the two programs remain similar, and variable names remain the same, matching variables based on name is suitable for generating assertions. On the other hand, matching variables based on behaviour allows variables that have been renamed to be matched, and corresponding assertions to be automatically generated. The technique for matching variables based on behaviour is presented in Section 5.2.2.

If a variable in the reference program cannot be matched with the corresponding variable used in the development program, further assertions for these variables cannot be generated. In this case, the programs are scanned for code fragments that are similar and executed prior to defining the variables that could not be matched. An assertion is generated for each corresponding variable, within the two similar code fragments, that influence the computation of the variables that could not be matched. This approach, presented in Section 5.2.3, identifies whether the variables that could not be matched contain different values due to code executed before or after the similar code fragments. If the assertions generated in the similar code fragments differ, the automatic approach can resume and continue to generate assertions that trace the error back to the origin of the fault.

Assertions are only generated for variables that reside in the same functions as the two starting variables. As described in Section 5.1.2, it is not practical to process more than one function at a time. If the value of a variable is defined by a definition that exists before the current function is called, or within a function called from the current function, special assertions are created, namely incoming parameter and function assertions. These assertions, as discussed in Section 5.1.2, are used to determine if the error flows into the current function, or out of a function called within the current function.

Once an assertion has been created for each variable that may influence the computation of the starting variables, the two programs are run to completion, and the assertion results are analyzed. As discussed in Section 5.1.1.1, the relationships between the generated assertions are maintained in a rooted directed acyclic graph where each
node represents an assertion. Child nodes represent assertions that test the inputs to the computation of the assertion represented by the parent node. This graph, called the Assertion Graph, is used to discover the assertions that identify where the two programs first begin to differ.

If the failed assertion is an incoming parameter assertion, the programs must differ before the current function is called. If the failed assertion is a function assertion, the programs first begins to differ within the called function. To continue to trace the flow of an error and identify where the two programs first being to differ, further assertions need to be generated in different functions. The user is prompted to confirm the functions in which to continue constructing assertions and the definitions from which to start. This approach is presented in Section 5.1.2.
Figure 5-1: Automatic Relative Debugging approach.
5.1.1 Generating Intraprocedural Assertions with Static Analysis

To automate the construction of assertions, the data flow information of the two programs is utilized. In a manner very similar to the manual approach, the data flow that leads to each erroneous variable is followed in order to locate where the two programs first begin to diverge. To re-iterate the manual approach, if the output of a computation is reported to be incorrect, the user would define additional assertions to follow the inputs to the faulty computation – typically by finding the definition points of the variables used in the computation.

The automatic approach starts from two variables that should contain the same value at a particular location in the two programs under consideration. The definitions that lead to these two variables are then located and an assertion is automatically created. Further assertions are automatically generated for the variables that are used in the computation of the located definitions.

For example, Figure 5-2 shows the assertions that are automatically generated from variable \( x \), on the last line of each program, as the starting point. By using the data flow information, the approach determines that \( x \) has one definition and subsequently creates one assertion. The variables used on the right hand side expression of this definition, \( a \) and \( b \), are then identified and further assertions are created. In this case, the variable \( a \) has one possible definition, resulting in one assertion. Similarly, two assertions are created for the variable \( b \).

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 3 )</td>
<td>( a = 3 )</td>
</tr>
<tr>
<td>( b = 2 )</td>
<td>( b = 3 )</td>
</tr>
<tr>
<td>( b = b * 2 )</td>
<td>( b = b * 2 )</td>
</tr>
<tr>
<td>( x = a + b )</td>
<td>( x = a + b )</td>
</tr>
<tr>
<td>print ( x )</td>
<td>print ( x )</td>
</tr>
</tbody>
</table>

← assertion 1: \( x \) →
← assertion 2: \( a \) →
← assertion 3: \( b \) →
← assertion 4: \( b \) →

Figure 5-2: Automatically generated assertions.
The manual approach for creating assertions usually continues by a process of refinement until the faulty section of code is localized. The user will typically create a small number of assertions to identify the set of variables that contain contrasting values when the two programs are run. This approach is efficient because the user can discard the set of variables that are correct, and requires that further assertions are only created across the set of variables that contain contrasting values. Hence, the process of refinement prevents the user from following and creating assertions for variables that are correct and are not influencing the incorrect result.

The automatic approach, on the other hand, is quick and can therefore create assertions across a large number of variables. The generated assertions however, need to be restricted to the current function for two reasons:

- It becomes difficult to trace variables across function boundaries. The problem is discussed in Section 5.1.2, along with the solution. In summary, the function may have been called from several places in the program which makes it difficult to trace the erroneous variables.

- Even though a large number of assertions can be generated quickly, the time required to execute the programs and perform the comparisons can take a long time, especially in large computational programs that contains iterative code. Hence, it is best to follow an approach similar to the manual approach and generate a limited number of assertions, run the programs to perform the comparisons, and generate further assertions for the faulty set of variables only.

5.1.1 Assertion Graph

The relationships between the assertions that are generated automatically are represented by the assertion graph. The assertion graph is a rooted directed acyclic graph where each node represents an assertion. Child nodes represent assertions that test the inputs to the computation of the assertion represented by the parent node.

Not only is the assertion graph used to identify certain properties about the generated assertions and automate the process of Relative Debugging (as discussed in the fol-
loring sections), it is also a useful visualization tool for users. The assertion graph clearly shows the relationship between generated assertions and highlights the flow of errors through the program. This information allows the user to quickly determine where their attention should be focused in order to correct the error.

As an example, Figure 5-3 shows the assertion graph for the assertions generated in Figure 5-2. The assertion graph shows that assertion 1, generated for variable x, has two inputs. The inputs, a and b, are tested by assertions 2 and 3 respectively and, assertion 4 tests the value of b which is an input to the computation tested by assertion 3. The shaded nodes in the graph represent assertions that have failed and show the flow of errors through the program. Hence, the user can quickly identify that the leaf node, representing assertion 4, is the origin of the error and requires further investigation.

Figure 5-3: Assertion graph for the assertions generated in Figure 5-2.
5.1.1.2 Multiple Definitions

Tracing the data flow in the manner discussed above allows assertions to be generated for very simple programs, but does not scale to programs with *slightly* more complex control and data flows. In particular, an assertion cannot be generated if, in either program, the value of a variable might have been assigned by one of a number of possible definitions. This situation occurs when different execution paths may be taken prior to executing the statement containing the variable of interest. The problem exists because an individual definition from program 1 cannot be safely matched with the corresponding definition in program 2.

For example, if the (modified but semantically equivalent) programs in either Figure 5-5 or Figure 5-6 are compared with the (original) program in Figure 5-4, the use of variable x, located on the last line of each program, has two possible definitions. We cannot safely determine that the definition of x, constructed with expr1 in program 1, should be matched with the definitions of x also constructed with expr1 in programs 2 or 3.

<table>
<thead>
<tr>
<th>If pred</th>
<th>If NOT pred</th>
<th>x = expr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = expr1</td>
<td>x = expr2</td>
<td>if pred</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td>x = expr1</td>
</tr>
<tr>
<td>x = expr2</td>
<td>x = expr1</td>
<td>end-if</td>
</tr>
<tr>
<td>end-if</td>
<td>end-if</td>
<td>Øx</td>
</tr>
<tr>
<td>Øx</td>
<td>Øx</td>
<td>use x</td>
</tr>
</tbody>
</table>

*Figure 5-4: Multiple definitions - program 1.*

<table>
<thead>
<tr>
<th>If NOT pred</th>
<th>x = expr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = expr2</td>
<td>if pred</td>
</tr>
<tr>
<td>else</td>
<td>x = expr1</td>
</tr>
<tr>
<td>x = expr1</td>
<td>end-if</td>
</tr>
<tr>
<td>end-if</td>
<td>Øx</td>
</tr>
<tr>
<td>Øx</td>
<td>use x</td>
</tr>
</tbody>
</table>

*Figure 5-5: Multiple definitions - program 2.*

<table>
<thead>
<tr>
<th>x = expr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>if pred</td>
</tr>
<tr>
<td>x = expr1</td>
</tr>
<tr>
<td>end-if</td>
</tr>
<tr>
<td>Øx</td>
</tr>
<tr>
<td>use x</td>
</tr>
</tbody>
</table>

*Figure 5-6: Multiple definitions - program 3.*

To eliminate the ambiguity when several definitions exist, we place the programs in Static Single Assignment (SSA) form [33, 39, 68]. When a program is in SSA form special Ø-functions are inserted at the point where the definitions that lead to each variable use converge in the control flow. Hence, SSA form ensures that only one definition may ever reach a variable’s use. This property eliminates the problem of matching multiple definitions and allows one assertion to be placed at the point
where Ω-functions have been inserted. In Figure 5-4, Figure 5-5 and Figure 5-6, for example, one assertion for \( x \) is placed outside the if statements, at the points illustrated by the SSA Ωx-functions. Consequently, a region of code that is producing incorrect values is identified rather than individual statements.

### 5.1.1.3 Control Variables

A variable in the evolved program may be assigned a conflicting value because an alternate control path is executed. Using the data flow properties of two programs to construct assertions, as outlined above, is not adequate if the variables used in the defining expressions are not among the variables influencing the executed control path.

For example, the variable VIP in Figure 5-7 is incorrectly assigned a different value in the second program and causes a different control path to be taken. The alternate control path computes a different, and obviously incorrect, value for the variable total. Hence, by simply considering the data flow leading to the computation of total, an assertion for the variable VIP would not be generated and the source of the error would go undetected.

<table>
<thead>
<tr>
<th>VIP = isVIP(cust.lastName)</th>
<th>VIP = isVIP(cust.fullName)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if VIP</td>
<td>if VIP</td>
</tr>
<tr>
<td>total = order + tax - disc</td>
<td>total = order + tax - disc</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>total = order + tax</td>
<td>total = order + tax</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>use total</td>
<td>use total</td>
</tr>
</tbody>
</table>

**Figure 5-7: Example showing the need to consider control variables when generating assertions.**

To avoid this problem the backward slice [130], as described in Section 4.4.1.2.1, is computed from each variable that is considered during the construction of assertions. Slicing allows assertions for variables that influence the definition of a variable but are not directly used in the computation to be generated e.g., variables used in predicates and loop invariants. Hence, in the above example, Autoguard generates an ad-
ditional assertion to compare the variable VIP although it is not in the data flow that leads to the computation of total.

### 5.1.1.4 False Positives

A false positive occurs when an assertion reports a failure because the compared values differ, but the values compared by the assertion actually should differ when the two programs are run. To illustrate false positives, Figure 5-8 shows two programs that compute the factorial of a requested number in a slightly different manner. The algorithm in program 1 computes the factorial by iterating from one through to the requested number, whereas program 2 computes the factorial by iterating from the number provided by the user down to one. Note that the assertions for both fact and n are placed outside the loop for reasons discussed in Section 5.1.1.2. If false positives are not considered, the programs will halt when assertion 2 is reached because the value of n will be different after control has exited the loop in each program. In particular, the variable n would contain 0 in program 1, and in program 2, would contain the number for which the factorial was computed, as requested by the user. Hence, assertion 2 is an example of a false positive because both programs, after they have run to completion, have correctly computed the same factorial value.

```plaintext
Program 1:
read reqFact
n = reqFact
fact = 1
while n > 0
    fact = fact * n
    n = n - 1
end-while
Øfact
Øn
print fact
```

```plaintext
Program 2:
read reqFact
n = 1
fact = n
while n <= reqFact
    fact = fact * n;
    n = n + 1;
end-while
Øfact
Øn
print fact
```

Figure 5-8: Example showing false positives.
To discover false positives, we allow the two programs to run to completion, performing all comparisons, and identify failed assertions for variables that are subsequently used in a computation proved correct by a successful assertion. Such assertions can easily be identified by performing a depth first traversal of the assertion graph; a failed assertion may be marked a false positive if any one of its ancestor assertions have passed.

For example, Figure 5-9 shows two programs that compute the same value for variable $x$ even though some intermediate computations differ. The figure also shows the assertion graph for the assertions that are generated across the two programs. The assertion graph has also been annotated with the values from programs 1 and 2 that would be used in the comparisons. Assertion 4, comparing the values of variable $b$, fails and is depicted by the shaded node in the assertion graph. Given that assertion 5, testing the variable $a$, succeeds after the programs have executed, any failed assertion that tests a variable used in the computation of variable $a$ can be marked as a false positive. Hence, assertion 4 can be marked as a false positive even though variable $b$ will contain different values after the two programs have executed.
Program 1

\[
\begin{align*}
a &= 10 \\
c &= 2 \\
b &= a + 10 \\
m &= b - 2 \\
x &= m * 2
\end{align*}
\]

\[\text{assert 5: } a \rightarrow \text{assert 3: } c \rightarrow \text{assert 4: } b \rightarrow \text{assert 2: } m \rightarrow \text{assert 1: } x \rightarrow\]

Program 2

\[
\begin{align*}
a &= 10 \\
c &= 2 \\
b &= a + 5 \\
m &= b + 3 \\
x &= m * c
\end{align*}
\]

Figure 5-9: Using the assertion graph to identify false positives.
5.1.1.5 Suspect Assertions

An assertion is marked as suspect when the variables compared by the assertion contain the same values after the two programs have run to completion, but the intermediate values assigned to the variables differed during execution. The assertion cannot be marked as failed because the variables compared by the assertion contain the same values after the programs have executed. It is useful to mark such assertions as suspect because they provide a useful insight into program execution and may help the user uncover the source of an error.

The programs in Figure 5-10, which compute the factorial of a given number, are used to demonstrate an instance where suspect assertions provide invaluable information that is used to identify the source of a problem. Figure 5-10 also shows two assertions and has been annotated to show the values assigned to the variables, compared by the assertions, during program execution. The two programs are expected to calculate and display the same result. However, program 1 computes and displays 24 where program 2 computes and displays 720 after they have both run to completion.

After the programs have run to completion, assertion 1 fails because the variable fact contains 24 in program 1 and 720 in program 2; assertion 2 succeeds because variable n contains 0 in both programs after they have run to completion. The source of the error is not highlighted because assertion 2, comparing the loop counter n, contains the value 0 in both programs after the programs have run to completion. The
variable \( n \) is actually the source of the error because the variable is initialized incorrectly and causes the loop to iterate an additional two times. Using suspect assertions, assertion 2 is marked as suspect because the variable in program 2 has been assigned two additional values.

### 5.1.2 Generating Interprocedural Assertions with Dynamic Analysis

As discussed in the previous section, static analysis is used to create assertions that trace the data flow of erroneous variables. Using static analysis in this manner allows the data flow properties of two functions to be easily traced, and variables to be matched such that assertions can be generated. The source of the error however, may not reside within the two functions\(^3\) that are being considered. The error may have occurred before the current function was called, or, during a function call. In either case, the static analysis approach, used to construct assertions within a function, is not powerful enough to trace and localize the fault. The approach is not suitable because a number of definitions, defined in different functions, may have been executed prior to the function being called. Furthermore, it is also possible that the current function is called from more than one function, and hence, the definitions that may have been executed prior to the function being called may be scattered throughout a number of different functions. Similarly, it is possible that multiple definitions are scattered throughout nested function calls. These situations result in multiple definitions that cannot be safely matched.

The approach adopted when multiple definitions exist within a single function is to place the two programs in SSA form and generate one assertion, as discussed in Section 5.1.1.2, that represents the set of definitions. Such an assertion is placed at the program point where the control paths that contain the multiple definitions converge. When the set of definitions are confined to a single function, the representative assertion will be created within the same function. This approach is not useful when multiple definitions reside in different functions. If the definitions are scattered throughout numerous functions, the representative assertion will be placed in a function where the control flow that exits from each of the functions containing the definitions

---

\(^3\) Function is used to refer to a function, method, or subroutine.
converges. This program location may not be located near the multiple definitions and will fail to localize and highlight the source of the error.

For example, assertion 1 in Figure 5-11 has been generated to compare the variable \( \text{res} \). Assertion 2 has been generated to compare the variable \( \text{tmp} \) used in the computation of \( \text{res} \). In this case, assertion 2 represents the definitions 1 and 2, which are located in two different functions. Hence, the representative assertion is not located near the source of the error and fails to localize the source of the fault. Furthermore, if we were to start constructing assertions for the variables used in definitions 1 and 2, further assertions would be created in the `compute` function.

<table>
<thead>
<tr>
<th>foo(a, b, c)</th>
<th>bar(x, y, z)</th>
<th>compute (n1, n2, res)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = a \times a )</td>
<td>( x = x \times 2 )</td>
<td>if ( (n1 / n2) &gt; 10 )</td>
</tr>
<tr>
<td>( b = b \times b )</td>
<td>( y = y \times 2 )</td>
<td>foo ( (n1, n2, \text{tmp}) )</td>
</tr>
<tr>
<td>( c = \text{sqrt}(a + b) )</td>
<td>( z = (x + y) / 2 )</td>
<td>else</td>
</tr>
<tr>
<td>res = ( \text{tmp} / 2 )</td>
<td></td>
<td>bar ( (n1, n2, \text{tmp}) ) end</td>
</tr>
</tbody>
</table>

\( \leftarrow \text{def 1: c} \rightarrow \) \( \leftarrow \text{def 2: z} \rightarrow \) \( \leftarrow \text{assert 2: tmp} \rightarrow \) \( \leftarrow \text{assert 1: res} \rightarrow \) \( \leftarrow \text{compute} \rightarrow \)

\( \text{Figure 5-11: Interprocedural assertions.} \)

A further reason that prevents definitions that reside in multiple functions from being considered is the approach adopted for matching renamed variables. The approach, as discussed in Section 5.2.2, identifies renamed variables by considering a function in the original program with a function in the evolved program. Hence, the approach cannot handle the situation where multiple definitions exist in multiple functions.
Dynamic analysis, along with some user interaction, is used to trace and localize a fault that does not reside in the current function. Dynamic analysis allows information about program execution to be collected and is used to trace the flow of errors across function boundaries, i.e., in and out of functions. The information allows the executed control path to be discovered and eliminates the functions, and definitions, that are not executed.

**5.1.2.1 Function Assertions**

The value of a variable may be assigned within a called function. If such a variable is passed to a function, or declared global, an assertion for the variable is placed before the function call, and a corresponding assertion is placed immediately after the call. These assertions are called *function-in* and *function-out assertions* respectively or, in general, *function assertions*.

If the function-out assertion fails, but the function-in assertion was successful, then the variable compared by the assertion first begins to differ within the called function. To localize the error, it is necessary to generate further assertions within the called function. On the other hand, if the function-in assertion fails, then the variable contains an incorrect value prior to the function call. In this case, there is no need to step into the function as the variable must have contained an incorrect value prior to the function call.

**5.1.2.2 Incoming Parameter Assertions**

The definition for a variable may not reside in the current function. If such a variable is passed, as a parameter, to the current function, or declared global, an assertion for the variable is placed at the beginning of the function. These assertions are called *incoming parameter assertions*.

When an incoming parameter assertion fails, the associated variable must have been assigned an incorrect value prior to invocation of the current function. To localize the error, further assertions for the faulty variable must be generated from the call that invoked the current function.
5.1.2.3 Selecting Continuing Definitions

When a function or incoming parameter assertion fails, further assertions need to be generated in order to trace and localize the source of the error. The additional assertions need to be generated in a function other than the one currently being processed. In particular, if a function assertion fails, additional assertions will need to be placed in the called function. Conversely, if an incoming parameter assertion fails, further assertions will need to be generated in the function that invoked the function containing the incoming parameter assertion. In either case, the program point from which to continue constructing assertions needs to be identified.

For the same reason that multiple definitions in the same function cannot be matched, as discussed in Section 5.1.1.2, it is not possible to safely match definitions in program 1 with the corresponding definition in program 2. The approach we present below presents a technique that identifies a possible match, based on the values assigned to the variables in each program during execution. The user is asked to confirm the identified match or, in the case where the match is incorrect, manually select the matching definitions from which to continue tracing the data flow and constructing assertions.

If a function assertion has failed, it is possible to continue from the return point in the called function, and for an incoming parameter assertion, to continue from the call that invoked the current function. However, this approach is restrictive in cases where the executed definition that leads to the function or incoming parameter assertion has been moved to a different function in the evolved version.

For example, if the construction of assertions was to continue from the callsite shown in Figure 5-12, no definitions for variable $x$ could be matched, preventing the construction of further assertions. The problem in this example is that an additional function has been introduced in the second version and does not contain the definition of $x$, as used in the starting function.
A better approach is to continue from the executed definition that leads to the function or incoming parameter assertion. Hence, in the example above, continuing from the definition executed prior to the call allows further assertions for the computation of $x$ to be generated. The approach also eliminates the ambiguity of multiple definitions and reduces the region that assertions will cover.

An incoming parameter or function assertion may have performed a number of comparisons during execution of the two programs. It is possible that the values compared by these assertions may be produced by different definitions. Some comparisons may have failed, some may have passed. In such cases, it is possible to select the first definition that fails.

The incoming parameter assertion in Figure 5-13 for example, would fail on every comparison. In this case, definition 1 represents the definition that provides a value for $x$ on the first invocation of the function containing the incoming parameter assertion. Hence, definition 1 provides a good point from which to continue tracing the data flow for $x$ and generating assertions.
A slightly more sophisticated approach is to identify if any comparisons passed, and then locate the next comparison that failed. This approach would prevent reaching definition 2 being selected in the following example.

Figure 5-13: Selecting the continuing definition – example 1.

Figure 5-14: Selecting the continuing definition – example 2.
As discussed at the start of this section, it is not possible to safely identify matching definitions. Hence, using the above technique a *possible*, or *best guess*, match is identified. The user is asked to confirm the possible match or, in the case where the match is incorrect, select the definitions from which to continue.

### 5.1.2.4 Call Sensitive Assertions

To trace the flow of errors when a function assertion fails, additional assertions are generated in the called function, as described in the previous section. These assertions should only perform the associated comparisons when the function is invoked from the call associated with the function assertion.

If no consideration is given to the call site, the assertions that are placed in a function will perform the associated comparisons every time the function is invoked in the two programs. Allowing the assertions to perform the associated comparisons each time the function is invoked may result in the comparison of variables that do not influence the variable(s) being traced. Performing the comparison each time a function is invoked, without regard for the callsite, has two disadvantages. Firstly, the results that are presented to the user may contain the results of irrelevant comparisons. In such cases, the user will need to sift out (the potentially large number of) values that are not relevant. The second disadvantage is that the irrelevant comparison results are likely to make the identification of the continuing definitions less accurate. In the event that the selected definition is not correct, the user must manually identify the continuing definition and is subsequently presented with the first disadvantage.

Figure 5-15 illustrates the problem; the three function assertions are created, and after the programs are executed, the function assertion for variable \( a \) fails. In this case, the value of \( a \) is the same prior to the function call so it is necessary to step into the called function and construct assertion 2.1. After the programs are executed again, assertion 2.1 has been executed twice, once for the invocation that passes the variable \( a \), and once for the invocation that passes the variable \( b \), even though we are only interested in the first invocation.
To prevent this situation, *call sensitive* assertions must be used. A call sensitive assertion ensures that the assertions placed within a function only performs the associated comparisons when invoked from a particular call site.

### 5.1.2.5 Iterative Code

Assertions may be generated in a function that is called many times from the same callsite e.g., the call to the function may reside within a loop, or the function may be recursively called. In the case where the error resides in the function, the error may not occur during the first invocation, but rather, commence during a later invocation of the function. For this reason, the assertion results must be analyzed to determine if the variable being traced first begins to differ during a particular invocation, and if so, the user can be notified that the source of the error has been identified, and no further assertions need to be generated.

If the assertion results from each invocation are not analyzed, the source of the error may be overlooked and further assertions will be generated. In particular, the technique for selecting the continuing definition, as outlined in Section 5.1.2.3, will iden-
tify the incoming parameter assertion as faulty and step into the calling function; the error will go undetected and the user will not be notified.

For example, Figure 5-16 shows two programs that iteratively call the function `foo` and one assertion that compares the value of the variable `val`. Figure 5-17 shows the results of the comparisons performed by the assertion during each invocation and reveals that the variable `val` starts to differ during the fourth invocation. If the user is notified that the error occurs during the fourth iteration, the technique for selecting the continuing definition will identify that the incoming parameter assertion failed, during the fifth iteration, and select the definition in the main function to continue from. The desired result is to notify the user that the value of `val` first begins to differ during the fourth invocation of `foo`.

<table>
<thead>
<tr>
<th>foo(count, val)</th>
<th>incoming param assert: val</th>
<th>foo(count, val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if count &lt; 3</td>
<td>val = val + 1</td>
<td>if count &lt; 5</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>val = val * 2</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>end-if</td>
<td>val = val + 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end-if</td>
</tr>
<tr>
<td>main</td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>i = 0</td>
<td></td>
<td>i = 0</td>
</tr>
<tr>
<td>num = 0</td>
<td>while i &lt; 5</td>
<td>num = 0</td>
</tr>
<tr>
<td></td>
<td>foo(i, num)</td>
<td>while i &lt; 10</td>
</tr>
<tr>
<td></td>
<td>i = i + 1</td>
<td>foo(i, num)</td>
</tr>
<tr>
<td></td>
<td>end-while</td>
<td>i = i + 1</td>
</tr>
</tbody>
</table>

Figure 5-16: Example showing iterative code.
To identify if a variable first begins to differ during function invocation, the assertion results from each function invocation must be analyzed. In particular, if an assertion testing a particular variable fails, but the incoming parameter assertion for the same variable succeeds, then the variable must begin to differ during the iteration. In this case, the user can be notified that the source of the error has been located and requires attention.

### 5.2 Unable to Generate Assertions when the Programs Differ

As discussed and listed in Chapter 2, a program may evolve for a large number of reasons. The changes that result in a new program version can range from a single, one line change, through to a number of large, complex changes. Special consideration is required when the changes prevent the automatic approach, as presented above, from tracing the flow of errors that lead back to the origin of a fault.

One particular change that requires special consideration is the renaming of variables. The automatic approach must identify variables that have been renamed, so that assertions can be created with the corresponding variables.

Special consideration is also required when the program code has been modified in such a way that the automatic technique can no longer generate assertions. This situation occurs when a variable in the original program cannot be matched with a variable in the evolved program. A typical change that may result in program code that cannot be matched occurs when an existing algorithm is replaced with a clearer
or more efficient algorithm. The replacement algorithm is likely to contain different computations that use a different set of variables and data flow.

The approach used to automate the construction of assertions mirrors the process that might be manually performed by the user. Similarly, the approach for handling renamed variables and different program code also mimics the manual approach.

When users reach the point where they cannot construct assertions because the two program versions have begun to differ, they will typically search back through the programs, attempting to identify similar pieces of code. If a portion of code that performs a similar function is identified, the user might create an assertion, at the end of the similar fragment of code, for each variable that is used at the point where the programs began to differ. This approach allows the user to determine whether the variables first begin to differ before or after the similar code. In the case that a variable begins to differ prior to the similar code, the user can proceed by creating further assertions to identify where the variables first begin to differ. If, on the other hand, a variable begins to differ after the fragment of code identified by the user, the source of the error must occur within this code. The region of code that assertions cannot be easily generated across may be small enough to use conventional debugging techniques to identify the source of the error. If the region is still quite large, or conventional debugging techniques are ineffective, the user may perform more complex analysis to identify suitable assertion points within the region and further narrow the scope of the error. The similar pieces of code are also used by the user to help identify corresponding variables that have been renamed.

5.2.1 Cloning for Automatic Relative Debugging

To automate this process and identify suitable program points from which to construct further assertions, software clones are utilized. A clone is a region of software that is identical or similar to another. Cloning is a topic that has received recent attention and is being applied in a number of areas, including source code optimization, refactoring and maintenance [35, 115].
Rysselberghe and Demeyer [115] conducted empirical research to evaluate and identify the cloning techniques that are most suited for individual maintenance tasks. The research classified the techniques into the following three categories:

- **String Based.** Sequences of duplicated strings within the program(s) are located using basic string transformation and comparison algorithms. Examples include comparing calculated signatures per line [75], parameterized line matching [64] and text based comparisons [44].

- **Token Based.** The comparison is performed on a token representation of the program(s). The token approach allows an improved comparison algorithm to be used, but requires a lexer to transform the program into a stream of tokens. Examples include token based representations [76] and parameterized matching using suffix trees [26].

- **Parse Tree Based.** Pattern matching is performed on an abstract representation of the program(s) to find similar structures. The richness of such representations allow sophisticated comparison algorithms to be used. The name of this category is misleading as the associated techniques may use representations other than parse trees. For example, [30] uses an abstract syntax tree, [95] use an intermediate representation language, [79] and [81] use program dependence graphs. An alternative and apt name for this category could be Abstract Representation Based.

A representative technique from each category was selected and evaluated for use with common maintenance tasks. Based on the results, the authors conclude that “the different clone detection techniques reported in the literature each have specific advantages compared to others. As such, each technique is more appropriate for a certain maintenance task.”

Burd and Bailey [35] also performed empirical research to identify which clone detection technique is best suited to assist with software maintenance, and, in particular, preventative maintenance. Five tools that use different techniques to identify clones were assessed.
The experiment involved using each tool on a medium sized application and comparing the identified clones. The results revealed that “each tool identified some clones that were not identified by any other tool and that each tool overlapped those that it identified with other tools. In all instances these overlaps were different. Only through using all the tools would it have been possible to identify the total set of clones.” In summary, “the results show that each tool has its own strengths and weaknesses and no single tool is able to identify all clones within the code.”

Hence, the successful application of cloning to automate Relative Debugging in the presence of changes will depend largely on the selected cloning technique. A further limitation that impacts the usefulness of current cloning techniques for Relative Debugging purposes is the inability to identify code that has been modified but remains semantically equivalent. For example, current cloning techniques cannot identify that the three different code segments in Figure 5-18 are semantically equivalent. Ideally, a cloning algorithm suited for Relative Debugging purposes would be able to detect sections of code that have undergone semantics preserving modifications, including those shown in Figure 5-18.

| x = 0       | if P then               | if NOT p then           |
|            | x = 1                   | x = 0                   |
| if P then   | X = 1                   | else                    |
| x = 1       | X = 0                   | x = 1                   |
| end-if      | end-if                  | end-if                  |
| y = x       | y = x                   | y = x                   |
| output y    | output y                | output y                |

Figure 5-18: Semantics preserving transformations not identified by current cloning techniques.

Wang et al proposed a technique in 1989 to identify semantics preserving transformations, such as those in Figure 5-18 [137]. The results however, were only applicable with a restricted language designed for research purposes and were not extended to handle commercial languages. Unfortunately, semantic equivalence is, in general, undecideable. Furthermore, the techniques need to evolve from being text or syntax based and consider semantics. With further research and development
however, the techniques will become more powerful and be able to identify a larger class of semantically equivalent modifications. This is indicated in [81] where a technique that detects modified duplicates is proposed.

As discussed in the following two sections, we have successfully used cloning to identify renamed variables and suitable program points from which to continue constructing assertions. The use of cloning has proven successful, as shown in Chapter 6, and should work well under many software evolution scenarios. Given the current restrictions however, it is not always possible to identify program points from which to continue generating assertions. Furthermore, the region of code at which the data flow begins to differ, and the program point(s) which we can automatically identify as suitable points which to continue from, may cover an excessively large region of code such that the system has not assisted the user to localize the error.

### 5.2.2 Renamed Variables

The variables in the original program version may be matched with the corresponding variables in the evolved version before assertions are generated across two functions. When a variable is used in a computation and needs to be traced, the matching variable is retrieved and used to construct the assertion. If a matching variable was not discovered, no assertions are generated and the automatic construction of assertions ceases at this point.

Variables can be matched by name or behaviour. Matching variables by name simply involves matching variables with the same name. Matching variables by behaviour involves identifying two variables that exhibit the same execution behaviour. To identify such variables software clones are used. Given that software clones identify pieces of code that exhibit the same execution behaviour, variables within the clones can also be matched.

The examples presented to this point have illustrated the construction of assertions based on matching variables by name. Figure 5-19 shows a small example where the variables are matched based on behaviour.
\( a = 10 \) \quad \text{assert 4: } a \rightarrow aVar = 1

---

**Clone 1**

\( b = 20 \) \quad \text{assert 5: } b \rightarrow bVar = 20

\( a = a \times a \) \quad \text{assert 2: } a \rightarrow aVar = aVar \times aVar

\( b = b + a \) \quad \text{assert 3: } b \rightarrow bVar = bVar + aVar

\( c = a + b \) \quad \text{assert 1: } c \rightarrow cVar = aVar + bVar

print c \quad \text{print cVar}

Figure 5-19: Generating assertions for renamed variables.

Matching variables by behaviour is only as strong as the cloning algorithm that is applied. The cloning algorithm may not discover equivalences and allow all variables to be matched. In this case, the automatic approach could allow the user to manually match the variables that could not be matched.

### 5.2.3 Different Data Flow

If the two program versions begin to differ such that variables cannot be matched and assertions cannot be generated, the point where the programs begin to look similar again is identified. As discussed above, software cloning is used to identify the fragments of the code that exhibit the same functionality and allow assertions to be constructed across the two programs.

The clone represents code that exhibits the same execution behaviour. An assertion, for each variable that is being traced, is placed at the end of the clone. The programs are then concurrently executed, and if these assertions pass, the error must reside between the program points where the programs began to differ, and the clone. On the other hand, if the assertions fail, the error must occur before the clone. In this case, the automatic construction of assertions can resume tracing the flow of the error.

For example, Figure 5-20 shows two programs that populate an array with the monthly sales average for a number of salesmen. The array is then sorted so that the averages can be displayed in order. Finally, the overall monthly sales average is
computed. The programs are the same except that the sorting algorithm, in the second program, has been replaced with the more efficient insertion algorithm.

In the example, assertions 1, 2 and 3 are automatically generated before the data flow differs, preventing further assertions being generated. At this stage, cloning is used to identify any suitable program points where the construction of assertions can resume. In this example, two clones are identified and highlighted in Figure 5-20. Clone 1 is identified as a suitable fragment of code to resume the construction of assertions. Assertions 4, 5, and 6 are subsequently created.
```c
void DisplayAverages()
{
    int monthlyAvg[SIZE];
    int i, j, temp;

    /* Init array with monthly averages. */
    for (i = 0; i < SIZE; i++)
    {
        read monthlySales;
        read monthlyTotal;
        monthlyAvg[i]=
           monthlyTotal / 
           monthlySales;
    }

    /* Sort with bubble sort. */
    for (i = (SIZE - 1);
        i >= 0;
        i--)
    {
        for (j = 1;
            j <= i;
            j++)
        {
            if (arr[j-1] > arr[j])
            {
                temp =
                   monthlyAvg[j-1];
                monthlyAvg[j-1] =
                   monthlyAvg [j];
                monthlyAvg[j] =
                   temp;
            }
        }
    }

    /* Display the monthly averages, from highest to lowest, and the overall average. */
    for (i = 0; i < SIZE; i++)
    {
        totalAvg +=monthlyAvg[i];
        assert 3
        print(monthlyAvg[i]);
    }

    totalAvg = totalAvg / SIZE;
    assert 2
    print (totalAvg);
}
```

**Figure 5-20: Generating assertions when the data flow differs.**
5.3 Automatic Relative Debugging Summary

We have presented an approach that automates the Relative Debugging methodology. The approach automatically identifies the data structures and program points that trace the flow of an error back to the origin of the fault. The approach requires little human intervention and eliminates the need for users to have a detailed knowledge of the two programs under consideration. The following contributions have been presented:

- An approach that allows matching data structures and program locations to be identified such that assertions can be automatically constructed.
- An approach that allows suitable data structures and program locations to be identified when the programs differ and matching data structures cannot be identified when tracing an error back through a program.
- An approach that allows renamed variables to be matched such that assertions can be constructed automatically.

5.4 Autoguard: An Automatic Relative Debugger

An automatic Relative Debugger, named Autoguard, has been developed using the techniques described in the previous section. This section commences with an outline of the high level architecture of Autoguard, followed by a discussion of the design issues, as discussed in the previous section, that required special consideration.

5.4.1 Architecture

An important use of Relative Debugging is the debugging of applications that have been ported from one language to another. To ensure Autoguard remains consistent with this use, and to avoid constructing front-end parsers for individual languages, we chose to process an intermediate language, namely the Microsoft .NET Intermediate Language [101].

A .NET program is contained within a portable executable (PE) file that contains compiler generated instructions defined by the Microsoft Intermediate Language (MSIL) [102]. Performing the required analysis on the intermediate languages al-
allows Autoguard to process programs regardless of the high level programming language translated by the compiler. In practice, this allows Autoguard to be used with a larger number of development languages i.e., any language that targets the Microsoft.NET framework. For research purposes, the framework provides a common, well defined platform in which our ideas and techniques can be evaluated.

Autoguard has been integrated with Microsoft Visual Studio .NET [98] using the Visual Studio Integration Program (VSIP) [99]. VSIP provides a framework in which the Visual Studio .NET architecture may be extended. In particular, the framework allows Autoguard to be hosted by Microsoft .NET and utilize the services, including the integrated development environment (IDE), offered by the environment.

Autoguard relies on DUCT and VSGuard [57]. Figure 5-21 shows how Autoguard, DUCT and VSGuard are integrated within the Microsoft Visual Studio .NET environment

![Figure 5-21: Extending Microsoft Visual Studio .NET with Autoguard.](image-url)
5.4.1.1 DUCT

Autoguard uses DUCT to locate variable definitions and generate assertions that trace the flow of errors. DUCT was extended to compute the SSA Ø-functions so that Autoguard can generate one representative assertion when multiple definitions for a variable exist. DUCT computes the required SSA Ø-functions using conventional techniques [21], and is discussed in more detail in Section 5.4.2.1.

5.4.1.2 VSGuard

VSGuard exposes a public interface that allows clients, such as Autoguard, to create assertions, start the Relative Debugger, and retrieve the comparison results associated with each assertion.

VSGuard also allows clients to register to receive event information. VSGuard will notify registered clients when assertions are executed and a comparison result is available, or when a line containing an assertion’s variable has been executed in either the original or evolved program.

The interface exposed by VSGuard is listed in Appendix A.

5.4.1.3 User Interface

Autoguard requires users to identify the variables, within the two programs, from which to start constructing assertions. Typically, they will select two output variables that are displaying conflicting values.

Using DUCT, as discussed in Chapter 4, Autoguard creates an assertion for each definition that leads to the user selected variable. Autoguard then considers the located definition(s) and generates assertions for the variables used in the right hand side expression of each definition. Autoguard iteratively applies this process and builds assertions for the set of variables, back to the start of the function, that influence the computation of the variable originally selected by the user.
Figure 5-22 shows the Autoguard Control Panel that displays the generated assertions. Assertions prefixed with UA represent user defined assertions, assertions prefixed with GA represented generated assertions. Assertions are displayed in a tree view to illustrate the parent / child relationship (as represented by the assertion graph).

![Autoguard Control Panel](image)

**Figure 5-22: Autoguard Control Panel.**

### 5.4.2 Technical Details

The following design aspects required special implementation consideration:

- Multiple definitions.
- Selecting continuing definitions.
- Call sensitive assertions.
- Cloning for Relative Debugging.

#### 5.4.2.1 Multiple Definitions

To eliminate the ambiguity when several definitions exist, the programs are placed in Static Single Assignment (SSA) form. SSA form ensures that only one definition may reach a variable’s use. SSA form allows one assertion to be generated that represents a set of definitions. Such an assertion is placed at the program point where the control paths, containing the multiple definitions, converge.

The representative assertion, representing the multiple definitions, retrieves and compares the computed values of the multiple variables. However, if a variable is redefined after the definition and before the representative assertion is reached then the comparison will be incorrect because the value will be overridden. For example, Figure 5-23 shows a simple program and the assertions generated for variables $x$ and $b$. The representative assertion for $b$ would capture the incorrect value for $b$ if the
true branch of the conditional was executed. In particular, the value computed by
definition of \( b \) after the definition of \( x \) would be captured. This is incorrect because
the value used in the computation of \( x \) is required.

```
a = ...
if P
    b = b - 1
    x = a + b
    b = ...
else
    b = b + 1
    x = a * b
end-if
print x
```

**Figure 5-23: Overridden assertion.**

```
a = ...
if P
    b = b - 1
    x = a + b
    b = ...
else
    b = b + 1
    x = a * b
end-if
print x
```

**Figure 5-24: The need for breakpoints to retrieve the correct values.**
To eliminate this problem, a *breakpoint* is placed on each definition that defines the value that should be compared at the representative assertion. A breakpoint is generated by creating a VSGuard assertion and providing the same details for both variables. That is, the program, variable name, and line number will be the same, rather than specifying variable details from two different programs. The breakpoint is used to capture and record the value of a variable, after the definition has executed, and is subsequently used in the comparison of the representative assertion. Figure 5-24 shows the breakpoints that would be created to retrieve the values to be compared by the representative assertion.

### 5.4.2.2 Selecting Continuing Definitions

Additional assertions need to be constructed when a function or incoming parameter assertion fails. In either case, the assertions that trace the erroneous variable(s) will be constructed in a function other than the one currently being processed. In particular, if a function assertion fails, additional assertions will need to be placed in the called function. Conversely, if an incoming parameter assertion fails, further assertions will need to be generated in the function that invoked the function containing the incoming parameter assertion.

For the reasons discussed in Section 5.1.2.3, the definitions need to be determined from which one might continue constructing assertions. To achieve this, breakpoints are associated with each function and incoming parameter assertion. For function assertions, a breakpoint is placed on each definition, in the called function, that may define the value that the traced variable contains once the function has returned to the callsite. For incoming parameter assertions, a breakpoint is placed on each definition that may define the value of the variable on entry to the called function. For example, Figure 5-25 shows the breakpoints associated with a function assertion and an incoming parameter assertion.
foo(x)

bar(x)

bar(refParam)
  if cond
    refParam = ...
  else
    refParam = ...
  end-if

main()
  if p
    x = ...
  else
    x = ...
  end-if

foo(x)

print x

Figure 5-25: Using breakpoints to select continuing definitions.
The breakpoint records that a particular definition has been executed and the value computed by the executed definition. The breakpoints provide the information required to select a definition from which to continue constructing assertions. In particular, the ordered list of definitions (captured by the breakpoints) is scanned to identify the first definition in program 1 and program 2 that produce the same values. The next definition in program 1 and program 2 that produces contrasting values is selected as the potential definition from which to continue.

The user is asked the user to confirm the selected definitions. If the user identifies that the selected definitions do not correspond, they must nominate the definitions from the list of definitions that were executed.

5.4.2.3 Call Sensitive Assertions

As detailed in Section 5.1.2.4, call sensitive assertions ensure that assertions placed within a function only perform the comparisons when the function is invoked from a particular call site. This ensures that the assertions only compare the values of variables being traced into a function, and eliminates the comparison of arbitrary variables that are passed from other calls to the function.

To implement call sensitive assertions, two breakpoints are created at the callsite. One breakpoint is placed prior to the call, the other is placed immediately after the call. When the breakpoint prior to the call is reached, the assertions placed in the function about to be called are enabled. When the breakpoint that is placed after the call is reached, the assertions in the called function are disabled. An assertion only performs the associated comparison when it is enabled.

5.4.2.4 Cloning for Relative Debugging

Section 5.2.1 explains that the use of cloning to identify renamed variables and different data flow is largely dependent on the selected cloning technique, and that no one technique currently identifies the exhaustive set of clones combined by all techniques. Furthermore, each technique identifies a unique subset of clones that is not identified by any other techniques. We have implemented and applied two cloning techniques for use with automatic Relative Debugging.
5.4.2.4.1 Parameterized Matching

The first technique that we have applied to Relative Debugging was proposed by Baker in 1992 [26, 27]. The technique is a text based approach that locates maximal sections of code that are exact matches or parameterized matches, referred to as p-matches. Exact matches, as the name suggests, are sections of code that are exactly the same. Parameterized matches are sections of code that are the same except for identifier names e.g., variables and constants.

Matches are discovered by an efficient algorithm based on a specialized data structure called the parameterized suffix tree. A parameterized suffix tree is a generalization of a suffix tree [97, 131], but extended to encode the parameters in such a way that matches can be identified. In particular, each statement is transformed into a parameterized suffix tree where the first occurrence of each parameter is replaced with a 0 and, each subsequent occurrence of the parameter is replaced by the digit representing the distance from the previous occurrence of the same parameter. A parameterized match exists if there is a one-to-one function that maps the set of parameters in one string onto the set of parameters in another string.

This technique is extremely efficient as reported by numerous experiments conducted by Baker [26, 27]. One such experiment discovered 5550 duplicates with a length greater than 30 lines in 7 minutes in C programs containing more than 1 million lines of source code, which included 20% of the code [26]. The downside of the technique, that limits its applicability for Relative Debugging, is that the approach is text based and fails to identify clones when statements have been re-ordered, but where the semantics of the two programs remain the same.

5.4.2.4.2 Slicing with Program Dependence Graphs

The second cloning technique that we have applied to Relative Debugging constructs a Program Dependence Graph (PDG) for each function and performs program slicing to find isomorphic PDG subgraphs that represent clones. This approach was first proposed in 2001 by Komondoor and Horwitz [79] to identify and extract clones into a new function and reduce the maintenance overhead involved with duplicated code.
The technique has been reported successful on C programs ranging from a few thousand lines of code up to 12,000 lines of code.

A PDG [49] represents a function where nodes depict program statements and predicates, while edges reflect the control and data dependencies between the statements. The PDG representation encodes the control and data dependencies while removing the arbitrary ordering of statements in the source program. This property allows the approach to find non-contiguous clones i.e., clones whose statements have been reordered and do not occur as contiguous text in the program.

A combination of forward and backward slicing is used to find isomorphic subgraphs in the PDGs. The slicing algorithm identifies isomorphic subgraphs by traversing the PDG and matching nodes based 1) on the syntactic structure of the statement/predicate represented by a node, and 2) the control and data dependencies of the node.

5.4.2.4.3 Cloning and Intermediate Languages

To allow automatic Relative Debugging to be performed on programs that have been ported to another language we have chosen to process an intermediate language, namely the Microsoft .NET Intermediate Language (MSIL). Performing cloning on the intermediate code allows clones to be identified between different source languages. The approach, however, is highly dependent on the compiler and the manner in which the source code is translated to the intermediate representation.

The first approach, based on parameterized suffix trees, identifies duplicated code based on the textual representation of the intermediate code produced by the compiler for the source language. Baker [28] has applied the techniques on Java bytecode with reasonable success. The experiment involved comparing the bytecode produced from two Java programs. In our case however, the intermediate code may be produced by compilers that translate different source programs using different rules and principles. The result is intermediate code that is textually different and hence, the number of clones that will be identified by the technique may be limited.
The slicing approach proves a better match for Relative Debugging purposes. It can identify clones where the intermediate code has been translated by different compilers and the arbitrary order of statements may differ. The technique is demonstrated by the case studies found in the following chapter.

5.5 Summary

This chapter has presented the techniques that allow Relative Debugging to be automated and so enhance the paradigm. The following chapter applies the techniques in three case studies and to demonstrate that automatic Relative Debugging reduces the need for users to have a detailed knowledge of the programs under consideration, thereby improving productivity, and has a significant impact on current debugging practices.
6 Case Studies

This chapter presents three case studies that demonstrate the application of automatic Relative Debugging to localize faults with little human intervention. The case studies have been organized to clearly highlight the individual contributions of the research outlined in this thesis.

The first case study involved porting an existing VB program to the .NET framework using C#. The contributions that are highlighted by this case study are 1) the ability to automatically construct assertions in order to identify the origin of an error, and 2) the ability to identify variables that have been renamed and to construct assertions accordingly.

The second case study demonstrates the ability to create assertions that trace an error through a number of function calls. This case study involved porting a scientific application, written in Fortran.net, to the .NET framework using C++. The data flow of the program is complex and difficult to trace. The automatic approach definitely proved a success in this case.

The last case study illustrates the techniques that allow automatic Relative Debugging to resynchronize, and continue constructing assertions, after the data flow has differed. Another contribution that is highlighted by this case study is call sensitive assertions. The call sensitive assertions remove unrelated values, allowing the user to concentrate on the source of the error.

In addition, the first two case studies involved porting the original application to a different language. A further contribution that is therefore highlighted by these two case studies is the ability to construct assertions and perform automatic Relative Debugging with programs that are written in two different source languages.
6.1 Case Study 1: Convert Base Program

This case study demonstrates how Autoguard may be used to locate errors that are introduced when a program is ported to a different programming language. The techniques that allow assertions to be automatically generated are illustrated. In addition, the ability to identify renamed variables and construct assertions accordingly is also highlighted.

In particular, the case study shows how we were able to locate and resolve the errors efficiently without a detailed knowledge of the underlying algorithm. Furthermore, it also shows how Autoguard discovered an error that would have otherwise gone undetected without thorough testing, but could have caused the application to terminate unexpectedly during operational use in the future.

The program used in this case study accepts a number expressed in base 2, 8, 10 or 16, via a graphical interface, and converts it to the equivalent number in base 2, 8, 10 or 16 as requested by the user. The original program, freely available from [3], was written in VB. For this case study, we chose to port the program to the .NET framework using C# and to standardize variable names using the Hungarian [121] naming convention.

Hungarian Notation, first suggested by Charles Simonyi [121] during the 1970s, provides a standard approach for naming identifiers (such as variables), constants, functions and classes, so that the comprehensibility of the program is increased. The convention states that an identifier should be meaningfully named, using mixed case, and be prefixed with one or more letters that reflect the type and scope of the identifier. For example, a numeric variable that represents a person’s age may be named nPersonAge. strUserName is another example of a variable named using the convention and, in this case, is a string representing a user’s name. A variable name may also be prefixed with a character to indicate the scope. For example, the name g_nYear represents a global numeric variable, and m_nAmount represents a numeric variable that is a member of an object.
Figure 6-1 shows the variable names in the original program alongside the corresponding renamed variables in the ported program. Figure 6-2 shows the variables that Autoguard matched using the cloning techniques described in Section 5.2.2. As shown in Figure 6-2, Autoguard failed to match six variables because the cloning technique did not identify any clones containing these variables. The reason that all variable were not matched, even though the two programs are very similar, was because the programs were translated to the intermediate language using different compilers. The different compilers produced programs that contained different intermediate code with subtle differences that prevented all variables being matched. To proceed, we manually matched the remaining variables.

<table>
<thead>
<tr>
<th>Original Variable Names</th>
<th>Renamed Variable Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitValue</td>
<td>dblBitValue</td>
</tr>
<tr>
<td>CompareWith</td>
<td>strCompareWith</td>
</tr>
<tr>
<td>ConvertBase</td>
<td>strConvertBase</td>
</tr>
<tr>
<td>Counter</td>
<td>lngCounter</td>
</tr>
<tr>
<td>DecimalChars</td>
<td>dblDecimalChars</td>
</tr>
<tr>
<td>DecimalValue</td>
<td>dblDecimalValue</td>
</tr>
<tr>
<td>InputBase</td>
<td>InputBase</td>
</tr>
<tr>
<td>InputCharCount</td>
<td>lngInputCharCount</td>
</tr>
<tr>
<td>inputValue</td>
<td>InputValue</td>
</tr>
<tr>
<td>Output</td>
<td>strOutput</td>
</tr>
<tr>
<td>OutputBase</td>
<td>OutputBase</td>
</tr>
<tr>
<td>OutputCharCount</td>
<td>dblOutputCharCount</td>
</tr>
<tr>
<td>Pos</td>
<td>dblPos</td>
</tr>
<tr>
<td>TmpChar</td>
<td>strTmpChar</td>
</tr>
<tr>
<td>TempLen</td>
<td>intTmpLen</td>
</tr>
</tbody>
</table>

Figure 6-1: Autoguard case study 1 - renamed variables in ported program.
6.1.1 Test 1: Converting Base 16 to Base 10

**Problem 1.** The first problem we encountered was that the ported program failed to display a result when converting a number from base 16 to base 10. To determine the source of the problem we located the functions, in the original and ported program, that perform the conversion. We then instructed Autoguard to automatically generate the assertions from the return variables within these functions. The generated assertions are displayed in the Autoguard Control Panel which can be seen in the bottom half of the screen illustrated in Figure 6-3. Figure 6-3 also shows the original VB .NET code and the ported C# code in the left and right source code panes respectively.
The source of the problem was quickly identified after Autoguard created 27 assertions and executed the programs. The variables `CompareWith` and `strCompareWith` are assigned strings that contain the characters that are valid for the input base. The strings are defined by extracting the first \( n \) numbers, where \( n \) represents the input base, from a constant string defined “01234…XYZ”. Assertion GA0009, illustrated in Figure 6-3, showed the values of `CompareValue` and `strCompareWith` differed, causing the ported function to return prematurely because the validation of the input value against `strCompareWith` failed. Inspection of the definition tested by assertion GA0009 in the ported code revealed that the computation of `strCompareWith` was incorrectly computed using the
String.Remove method to extract the valid characters from the front of the constant string. The problem was corrected by replacing the use of the String.Remove method with the String.Substring method. The incorrect statement was:

```
strCompareWith = csValidChars.Remove(0, (int) InputBase);
```

The statement was corrected as follows:

```
strCompareWith = csValidChars.Substring(0, (int) InputBase);
```

**Problem 2.** After correcting the initial problem the ported program displayed a value, but it differed from the value computed by the original program. Figure 6-4 shows the original program correctly converting the input value to 1044506 and Figure 6-5 shows the incorrect result, 974601, produced by the ported program.

![Figure 6-4: Autoguard case study 1 - converting base 16 to base 10 in the original program.](image)

![Figure 6-5: Autoguard case study 1 - error converting base 16 to base 10 in the ported program.](image)
We proceeded by re-executing Autoguard with the 27 assertions previously generated and noted that 4 assertions failed. Failed assertions GA0003 and GA0010, illustrated in Figure 6-6, identified a problem but did not influence the computation of the erroneous value. In fact, the assertions discovered a problem that probably would have gone undetected unless thorough and exhaustive testing had been performed.

The problem occurred within a validation stage where each character in the input string is tested to ensure that it is legal for the given input base. Assertion GA0003 showed the loop counters, Counter and lnrCounter, that iterate over the characters in the input string differed. In particular, Counter in the original program contained the value 6 on termination of the loop and indicates that the characters 1 through 5 in the input string were validated. In the ported program however, lnrCounter contains the value 5 after the loop has terminated and indicates that only the first 4 characters in the input string were validated. This meant that the last character in the input string was not validated and could cause the ported program to produce incorrect values, or, even worse, terminate unexpectedly if the last character in the input string was not valid for the input base. Assertion GA0010 supported this finding by showing that the last characters validated in the original and ported programs differed i.e., ‘A’ and ‘1’ respectively. The ported program was easily fixed by correcting the loop condition to iterate the correct number of times by using the less than equal comparison operator (\(\leq\)) instead of the less than operator (<).
Figure 6-6: Autoguard case study 1 – the second problem as identified by Autoguard
Problem 3. The remaining two failed assertions, shown in Figure 6-8, successfully identified the problem that caused the ported program to calculate and display an incorrect value. Assertion GA0021 showed that the variable `dblPos` contained a different value in the ported program.

The algorithm iteratively computes the conversion value by considering each character in the input string in turn. The value of each character is based on its position in a string array containing the valid characters for the input base. For example, the character ‘A’ in a hexadecimal number would be found at position 10 in the string array. Assertion GA0021 showed that the value for the last character considered (which was ‘F’ as shown by assertion GA0022) differed in the two programs and caused the ported program to use an incorrect value when iteratively computing the converted value. Figure 6-7 shows the values that were assigned to `dblPos` and that they differed (by one) throughout execution of the two programs.

Investigation revealed that the error occurred because string arrays are offset from 1 in VB .NET but are offset from 0 in C#. Hence, the value of a character in the ported program did not correspond to its position in the string array. To resolve the problem, the ported program was modified so that 1 was not subtracted from `lPos` after identifying the position of the character in the string array – this ensured the retrieved value correctly represented the value of the character. The incorrect statement was:

```
dblPos = strCompareWith.IndexOf(strTmpChar) - 1;
```

The statement was corrected as follows:

```
dblPos = strCompareWith.IndexOf(strTmpChar);
```

After correcting the above problems, the two programs produced the same values when converting values from base 16 to base 10.
Figure 6-7: Autoguard case study 1 - values captures by assertion GA0021.

Figure 6-8: Autoguard case study 1 - failed assertions indicating source of error.
6.1.2 Test 2: Converting Base 16 to Base 8

After further testing we discovered that the ported program computed conflicting values when converting numbers from base 16 to base 8. For example, Figure 6-9 shows the incorrect base 8 value computed for input value of base 16.

We ran Autoguard with the same 28 assertions that were generated earlier and found that a number of problems existed.

**Problem 4.** Assertion GA0014, shown in Figure 6-10, identified that the comparison for `dblOutputCharCount` failed because the value in the ported program was being truncated on line 469. It was thought that the problem could be corrected by removing the explicit conversion of the computed value (introduced for some reason, during the porting process) to type long before being assigned to `dblOutputCharCount`. The correction however, did not correct the problem because the variable, used in the loop condition on line 477, was being truncated before being used.

**Problem 5.** Further investigation revealed that the error occurred because the `while` loop on line 490, shown in the source code pane of Figure 6-10, was iterating too many times and not terminating as expected. In this particular case however, the failed assertions did not directly identify the source of the error.

The reason that Autoguard did not identify the problem was because assertion GA0012 which tested the value of the variable used in the loop condition, `dblBitValue`, was placed outside the loop (on line 490) for the reason mentioned in Section 5.1.1.2. Therefore, the value of `dblBitValue` was only compared once the loops had terminated and the values were obviously the same.
Figure 6-9: Autoguard case study 1 – error converting base 16 to base 8 in the ported program.

Figure 6-10: Autoguard case study 1 - using Autoguard to correct unhandled runtime exception.
Figure 6-10 shows that assertion GA0012 passed but has been marked as a suspect result. The question mark indicates that the variable contained the same value after the two programs ran to completion, but the values computed during execution differed. By clicking on assertion GA0012 in the control panel it could be seen, as shown in Figure 6-11, the list of values computed for \texttt{dblBitValue} during execution.

![Figure 6-11: Autoguard case study 1 - values computed for cuBitValue.](image)

The list of computed values showed that \texttt{dblBitValue} in the ported program was assigned an additional 371 different values before the variable was assigned a value of 0, which caused the loop to iterate an additional 371 times; C# was not rounding the computed value for \texttt{cuBitValue}, unlike VB .NET that was implicitly rounding the value. The correction required changing the definition on line 496 to explicitly perform the rounding, as follows:

\[
\texttt{cuBitValue} = \text{Math.Round(}\texttt{dblBitValue} \div \texttt{OutputBase});
\]
6.1.3 Case Study 1 Summary

This case study successfully demonstrated the use of Autoguard to locate errors that were introduced when a program was ported to a different programming language. In particular, Autoguard successfully identified five different errors that prevented correct values being calculated and displayed to the user. The only information required of the user was the corresponding names for six variables that could not be matched. The reason that Autoguard failed to automatically match these variables was because the program was ported to a different language, and the compiler emitted code differently, so that clones containing the unmatched variables could not be identified. The case study also highlighted the success of cloning to identify renamed variables.
6.2 Case Study 2: Shallow Water Equations

The second case study applies the interprocedural techniques, detailed in Chapter 5, to trace an error through functions calls, and to isolate such an error in programs designed to compute numerical solutions of the shallow water equations [116]. The shallow water equations “describe the motion of an incompressible fluid with a free surface, with the constraint that the horizontal scales of motion are much larger than the vertical. Although they use a very simple representation of the atmosphere, they do include the two types of horizontal wave motion important in more realistic global climate models, gravity and Rossby waves.” [9]. Figure 6-12 shows the variables computed by the model.

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potential energy</td>
</tr>
<tr>
<td>Kinetic energy</td>
</tr>
<tr>
<td>Total energy</td>
</tr>
<tr>
<td>Potential enstrophy</td>
</tr>
</tbody>
</table>

Figure 6-12: Autoguard case study 2 - variables computed by the shallow water equations.

The equations are commonly used for experiments with various model structures and numerical schemes. Hence, programs to solve the equations have been developed in a number of languages and architectures [9]. For this case study, we have ported an existing Fortran version to C#. Source code for the original Fortran and ported C# versions is listed in Appendix B. The erroneous lines in the ported C# version have also been highlighted in bold font in Appendix B.

The original Fortran version exercised a model size of 32 x 32 and a run length of 950 time steps. For development purposes, and so as to limit execution time, we reduced the model size to 4 x 4 with a run length of 15. The two programs also report a number of key values every 5 time steps.
After porting the original Fortran version to C#, a number of errors were witnessed. The correct result, produced by the Fortran version, can be seen in Figure 6-13. Figure 6-14 shows the output from the ported C# version and reveals that the computed values were incorrect after every 5 time steps. In particular, the C# version displayed ‘infinity’ which indicated that the computed values were too large to be stored in the declared variables.

<table>
<thead>
<tr>
<th>Shallow Water Simulation (original - fortran.net)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of points in the X direction</td>
</tr>
<tr>
<td>Number of points in the Y direction</td>
</tr>
<tr>
<td>Grid spacing in the X direction</td>
</tr>
<tr>
<td>Grid spacing in the Y direction</td>
</tr>
<tr>
<td>Time step</td>
</tr>
<tr>
<td>Time filter parameter</td>
</tr>
<tr>
<td>Number of time steps in run</td>
</tr>
<tr>
<td>Cycle number</td>
</tr>
<tr>
<td>Model time in days</td>
</tr>
<tr>
<td>Potential energy</td>
</tr>
<tr>
<td>Kinetic Energy</td>
</tr>
<tr>
<td>Total Energy</td>
</tr>
<tr>
<td>Pot. Enstrophy</td>
</tr>
<tr>
<td>Cycle number</td>
</tr>
<tr>
<td>Model time in days</td>
</tr>
<tr>
<td>Potential energy</td>
</tr>
<tr>
<td>Kinetic Energy</td>
</tr>
<tr>
<td>Total Energy</td>
</tr>
<tr>
<td>Pot. Enstrophy</td>
</tr>
<tr>
<td>Cycle number</td>
</tr>
<tr>
<td>Model time in days</td>
</tr>
<tr>
<td>Potential energy</td>
</tr>
<tr>
<td>Kinetic Energy</td>
</tr>
<tr>
<td>Total Energy</td>
</tr>
<tr>
<td>Pot. Enstrophy</td>
</tr>
</tbody>
</table>

***** End of Shallow *****

Figure 6-13: Autoguard case study 2 - original Fortran (correct) output.
To produce a .Net binary, the original Fortran version was compiled with the Salford Fortran.net compiler [4]. This step was straightforward and did not require any modification to the original code. The original Fortran code, however, was not structured with subroutines and contained a large number of goto statements to perform the required iterations. The ported version has been developed with structured programming techniques using functions and conventional looping constructs (as shown
in the source code listed in Appendix B). Hence, although the ported code looks quite different from the original, the data flow remains the same.

We commenced to debug the problem by focusing on the erroneous potential energy variable, \( ptot \).

### 6.2.1 Error 1: Incorrect Calculation

**Iteration 1.** We instructed Autoguard to generate the assertions from the statement that displayed the value of \( ptot \). Autoguard discovered that the value of \( ptot \) is computed in a function and subsequently generated two assertions. Figure 6-15 shows the two assertions, one that tests the value of \( ptot \) prior to the function call, and one assertion that tests the value after the function has executed.

![Figure 6-15: Autoguard case study 2 - iteration 1 assertions.](image)
Iteration 2. The results, after running the two programs, are also shown in Figure 6-16. Assertions GA0000 and GA0001 both failed. In this case, Autoguard would normally continue to generate assertions from the function call without stepping into the function. However, as the original version does not contain functions, Autoguard needed to step into the function in the ported program in order to correctly construct assertions across the two programs. The generated assertions, and results, are shown in Figure 6-16.

![Autoguard Control Panel](image)

Figure 6-16: Autoguard case study 2 - iteration 2 assertions.

Seven assertions had failed after the two programs were run. Assertions GA0004, GA0005, GA0009 and GA0010 compare the values of the loop counters used in the two programs. After investigating these assertions, it was determined that the variables actually contained the correct values and could be safely ignored. The assertions failed for two reasons. First, the Fortran.net arrays are indexed from one whereas the C# arrays are offset from zero. Hence, the loops need to iterate from different starting values. Second, the Fortran.net compiler translated the loops differently than the C# compiler. In particular, the Fortran.net compiler has produced code that increments the loop counter before testing the loop termination condition. In this case, the loop counter variable contains the value that is one greater than the loop condition. On the other hand, the C# compiler produced code that tests the loop condition before incrementing the loop variable. This situation can be seen by the values captured and compared by assertion GA0010, as shown in Figure 6-17.
These assertions were not flagged as false positives because the parent assertion, GA0003, had also failed. Hence, Autoguard could not determine if these assertions were influencing the erroneous computation of $ptot$, tested by GA0003 or not.

![Figure 6-17: Autoguard case study 2 – values captured by assertion GA0010.](image)

The remaining failed assertions, GA0002, GA0003 and GA0006, revealed the source of the error. The assertion graph, shown in Figure 6-19, shows the failed assertions after both programs were executed. The graph shows that GA0006, which compares the value of the incoming parameter $p$ is incorrect and influences the computation of $ptot$. Generally, Autoguard would proceed by generating assertions for the incoming parameter, $p$, to identify where the variable first begins to differ. However, in this case, Autoguard discovered that the point where $p$ first began to differ occurred during the third invocation of the function. In particular, Autoguard analyzed each invocation of the function and discovered that the incoming parameter $p$ contained the same value on the second invocation of the function, but differed on the third (as illustrated in Figure 6-18). This finding is also shown in Figure 6-20 which shows the state of the assertions graph after the second invocation of the function – in particular, the node representing assertion GA0006 is not highlighted and revealed that the incoming parameter contains the same value in both programs on the second invocation of the function.
Figure 6-18: Autoguard case study 2 – values captured by assertion GA0006.

Figure 6-19: Autoguard case study 2 – the assertion graph after both programs had completed..
After identifying that the programs began to differ during the second invocation of the function being processed, Autoguard continued to analyze the assertion results for this invocation. Using the assertion graph, as illustrated in Figure 6-20, Autoguard identified that the values compared by assertion GA0003 influenced the incorrect computation of the variables compared by assertion GA0002. In particular, GA0003 compared the values of the variable $PTOT$, and identified the point where the two programs first began to differ. As identified previously, the failed assertions GA0004, GA0005, GA0009 and GA0010 can be safely ignored because they compare the values of loop counters used in loops that are compiled and translated differently by the Fortran .net and C# compiler.
The computed values for PTOT started to differ during the second invocation of the function being processed. The computation, tested by GA003, in the Fortran.net version was:

\[ ptot = ptot + (p(i,j) - pmean)^2 \]

The corresponding computation in the ported version was:

\[ \text{PTOT} = (\text{PTOT} + (p[i,j] - pmean)) \times (\text{PTOT} + (p[i,j] - pmean)) \]

The following correction was required in the ported program:

\[ \text{PTOT} = \text{PTOT} + (p[i,j] - pmean) \times (p[i,j] - pmean) \]

After fixing the incorrect statement, the ported program no longer computed and displayed infinity for total energy.
6.2.2 Error 2: Incorrectly Initialized Loop Counter

Even though the ported program no longer computed infinity for potential energy, the value still differed from the potential energy value computed by the original Fortran.NET program. Figure 6-21 shows the values computed by the ported version (the original values can be seen in Figure 6-13).

<table>
<thead>
<tr>
<th>Shallow Water Simulation (C#)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of points in the X direction</td>
</tr>
<tr>
<td>Number of points in the Y direction</td>
</tr>
<tr>
<td>Grid spacing in the X direction</td>
</tr>
<tr>
<td>Grid spacing in the Y direction</td>
</tr>
<tr>
<td>Time step</td>
</tr>
<tr>
<td>Time filter parameter</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>Cycle number</td>
</tr>
<tr>
<td>Model time in days</td>
</tr>
<tr>
<td>Potential energy</td>
</tr>
<tr>
<td>Kinetic energy</td>
</tr>
<tr>
<td>Total energy</td>
</tr>
<tr>
<td>Pot. Enstrophy</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>Cycle number</td>
</tr>
<tr>
<td>Model time in days</td>
</tr>
<tr>
<td>Potential energy</td>
</tr>
<tr>
<td>Kinetic energy</td>
</tr>
<tr>
<td>Total energy</td>
</tr>
<tr>
<td>Pot. Enstrophy</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>Cycle number</td>
</tr>
<tr>
<td>Model time in days</td>
</tr>
<tr>
<td>Potential energy</td>
</tr>
<tr>
<td>Kinetic energy</td>
</tr>
<tr>
<td>Total energy</td>
</tr>
<tr>
<td>Pot. Enstrophy</td>
</tr>
</tbody>
</table>

***** End of Shallow *****

Figure 6-21: Autoguard case study 2 - errors in the ported Fortran program output.
**Iteration 1 & 2.** As with the first problem, we proceeded to debug the problem by concentrating on the erroneous potential energy variable, \( ptot \), and instructed Autoguard to generate the assertions from the statement that displayed the computed value. The same assertions used to locate problem 1 were generated, and, although the values in the ported program now differed, the same assertions also failed.

**Iteration 3.** Unlike the first problem, Autoguard did not identify the source of the error in the function that was processed during the second iteration. Autoguard proceeded by generating further assertions for the incoming parameter, \( P \), that was discovered incorrect on entry to the function being processed during iteration 2. The assertions for \( P \) were generated from the call that originally caused Autoguard to traverse into the function. It can be seen, as illustrated in Figure 6-22, that two assertions were created, namely GA0011 and GA0012, and were placed after the assertions generated and displayed in the control panel during the first iteration. This reflects that the assertions have been generated in the same function containing the function call that Autoguard stepped into and processed during the second iteration. The assertions generated during the second iteration have been collapsed in the figure for clarity.

![Autoguard Control Panel 1*](image)

**Figure 6-22: Autoguard case study 2 - Autoguard Control Panel showing errors during iteration 3.**

Only two assertions were generated for \( P \) because Autoguard discovered that the value of \( P \) is computed in a function. Subsequently, one assertion is generated to test the value of \( P \) prior to the function call, and one assertion is generated to test the value after the function has executed.
The two newly created assertions for $P$ failed after executing the two programs, as illustrated in Figure 6-22. Once again, as the original program does not contain functions, Autoguard had to step into the function in the ported program in order to match the definitions from the original program with definitions in the ported program. If both programs were structured with functions, there would have been no need to step into the function because the assertion for $P$, prior to the function call, also failed and indicated that the error occurred before the function call.

**Iteration 4.** Figure 6-23 shows the assertions that were generated for $P$ in the called function, and their results after running the two programs. Autoguard identified that $P$ began to differ during the first invocation of the function. The assertion graph for the function, after the first invocation, can be seen in Figure 6-24.

![Figure 6-23: Autoguard case study 2 - Autoguard Control Panel showing errors during iteration 4.](image-url)
As discovered during the first problem, failed assertions GA0016, GA0018 and GA0019 compare the values of loop counters used in the two programs, and can safely be ignored because loops are compiled and translated differently by the Fortran .net and C# compilers. Assertion GA0017 also compared the values assigned to the loop counters, however, the values differed in a way that did not reflect the slight differences produced by the two compilers. Figure 6-25 shows that the loop counter in the original version iterated from 1 through to 5, whereas the loop counter in the ported version was only iterating from 4 through to 5.

Figure 6-25: Autoguard case study 2 – values captured by assertion GA0017.
The statement responsible for the iteration in the Fortran .net source code was:

\[
\text{do 300 } j2=1, n
\]

The corresponding statement in the ported C# version was:

\[
\text{for (j2 = n; } \quad j2 \leq n; \quad j2++)
\]

The statement was incorrectly initializing the loop counter to the value of \( n \) rather than 0. The statement was corrected as follows:

\[
\text{for (j2 = 0; } \quad j2 \leq n; \quad j2++)
\]

After this correction, the two versions computed and displayed the same results.

### 6.2.3 Case Study 2 Summary

In this case study, Autoguard successfully discovered two non trivial errors that were introduced when the Shallow Water program was ported from Fortran to C#. The ported program was originally debugged using conventional Relative Debugging techniques with little success. The conventional technique was unsuccessful because it was difficult to trace and match the dataflow in the Fortran and C# versions. Autoguard, on the other hand, generated a number of useful assertions that quickly uncovered the source of the two problems. The first error was discovered within two iterations, the second error was located within four iterations.
6.3 Case Study 3: PREDICT – Satellite Tracking and Orbital Predictions

The third case study demonstrates the implementation of two individual contributions presented in this thesis. First, the case study shows the use of cloning techniques to identify code that is similar between the two programs (see Section 5.4.2.4). This information is used by Autoguard to find suitable data structures and programs points from which to continue constructing assertions after the programs have differed. The second contribution highlighted by this case study is call sensitive assertions (see Section 5.4.2.3). Call sensitive assertions remove unrelated values, allowing the user to concentrate on the source of the error.

The program used in this case study, PREDICT [7], is an open-source satellite tracking and orbital prediction program developed by John A. Magliacane. PREDICT provides the ability to track satellites as they orbit the earth as well as predict when a satellite is expected to come within range of a ground station. The prediction can also identify the upcoming passes that might be optically visible to a ground station. In addition, PREDICT tracks the positions of the Sun and Moon, and calculates the Doppler shift and path loss calculations for analyzing radio communication paths between satellites and earth-based ground stations. PREDICT is freely available under GNU General Public License (GPL) terms and conditions.

A number of organizations are currently using PREDICT for satellite tracking/orbital analysis applications. Organizations that have reported using PREDICT include:

- NASA: Goddard Spaceflight Center, Greenbelt, Maryland, USA.
- US Naval Research Laboratory, Washington, DC, USA.
- Stanford University's Space System Development Laboratory (SSDL), Stanford, USA.

PREDICT tracks and predicts passes of satellites based on the location of the ground station, the current date and time, and orbital data for each satellite of interest. The orbital data is used by the standard mathematical models of spacecraft orbits in order to calculate the position and velocity of near-earth and deep-space satellites. A near-
space satellite takes less than 225 minutes to perform a complete orbit of earth, where a deep-space satellite takes longer than 225 minutes. Based on the orbital data, five mathematical models [67] to calculate a satellite’s position and velocity are available:

- **SGP**: This model is used for near earth satellites.
- **SGP4**: This model is used for near earth satellites.
- **SDP4**: This model is an extension of SGP4 for deep space satellites.
- **SGP8**: This model is used for near earth satellites.
- **SDP8**: This model is an extension of SGP8 for deep space satellites.

According to [67] “The SGP8 and SDP8 models have the same gravitational and atmospheric models as SGP4 and SDP4, although the form of the solution equations is quite different. Additionally, SGP8 and SDP8 use a ballistic coefficient (B term) in the drag equations rather than the B* drag term.” PREDICT currently uses the SGP4 model for near-earth satellites, and the SDP4 model for deep-space satellites.

PREDICT has been released for Linux, DOS, and Sharp Zaurus. The Linux version has additional functionality that allows it to operate as a socket-based server and provide real-time tracking and orbital prediction information to client applications. A number of graphical clients that use PREDICT to track and display satellites are currently available.

For this case study, we ported the DOS based version, written in C++, to the .NET framework and replaced the SGP4 model with the SGP8 model. We also added the socket-based server functionality, as found in the Linux version, so that clients can retrieve satellite tracking and prediction information as required. The conversion was straightforward, requiring only two small changes.

The first change involved replacing use of the curses library [1], which provides character based functions to manipulate the terminal’s display, with an equivalent .NET version. The second modification was required with the introduction of the SGP8 algorithm. The SGP8 algorithm utilizes a different set of data structures from the SGP4 algorithm. The data structures contain similar information but have been
named slightly differently and/or extended to include additional information. Instead of modifying the SGP8 algorithm to use the existing data structures, we decided to create a function that maps the values from the data structures used by the SGP8 algorithm to the existing data structures and vice-versa. This approach did not require modifying the complex SGP8 algorithm and avoided the high possibility of introducing errors.

By visually comparing the original program with the ported program, it was immediately obvious that the ported version was computing the information incorrectly for each satellite. Figure 6-26 shows the original DOS version and the information computed for each satellite. Figure 6-28 shows the contrasting, and obviously incorrect, information as computed by the ported version. To illustrate the extent of the error, we connected a graphical client, gsat [7], to each version and visually inspected the computed locations of several satellites. Figure 6-27 shows the satellite named OSCAR-11 in the correct position, near the Caspian Sea in the USSR, as computed by the original version. Figure 6-29 shows the (incorrect) position, above Brazil in South America, of OSCAR-11 according to the ported program.

![Figure 6-26: Autoguard case study 3 - original PREDICT.](image-url)
Figure 6-27: Autoguard case study 3 – correct position of the OSCAR-11 satellite.

Figure 6-28: Autoguard case study 3 - ported PREDICT.
To debug the problem with Autoguard, two .NET binaries were required. To recompile the original DOS version, containing the SGP4 algorithm, for the .NET framework required replacing use of the curses library with an equivalent .NET counterpart. This modification was straightforward and allowed the original DOS version to be recompiled and run within the .NET framework. After confirming that the recompiled DOS version, running within the .NET framework, produced the same results as the original, we utilized this version with Autoguard to isolate the fault in the ported program.

Before proceeding to debug the problem, we modified the two programs and hard coded the current date and time. This was done to ensure that calculations that are dependent on the current date and/or time use the same values in the two programs.

To debug the problem we focused our attention on the erroneous latitude of one satellite, OSCAR-11. We located, in the SingleTracking function, the print state-
ment that displays the value of the latitude variable, sat_lat. The Single-Tracking function periodically computes and displays information for a selected satellite.

The results are shown in Figure 6-30 and explained below.

Figure 6-30: Autoguard case study 3 – Autoguard results for iterations 1 through to 5.

**Iteration 1.** We selected the variable representing the satellite’s latitude, sat_lat, in the SingleTrack function and instructed Autoguard to construct assertions that trace the error. Autoguard discovered that the value of sat_lat was defined in a function, Calc, and generated two assertions, one before the call, and one after.

**Iteration 2.** After Autoguard ran the two programs, assertion GA0003 indicated that the value of sat_lat prior the call to Calc was correct. On the other hand, assertion GA0004 revealed that the value of sat_lat contained a contrasting value after the call to Calc. This indicates that the variable is assigned an incorrect value in Calc. Hence, Autoguard generated further assertions that traced sat_lat into function Calc.
Assertion GA0004 also illustrates the advantage of call sensitive assertions. Call sensitive assertions, as discussed in Section 5.1.2.4, allows the associated comparisons to be performed when the function containing the assertions is invoked from a particular call. This allows Autoguard to only perform the comparisons when a variable of interest is passed by parameter to the function. Hence, the comparisons of variables that do not influence the result of the variable being traced are eliminated. For example, Figure 6-31 shows the value captured by assertion GA0004 with call sensitive assertions enabled. In contrast, Figure 6-32 shows the additional values that are captured, by the same assertion, if call sensitive assertions are disabled.

Figure 6-31: Autoguard case study 3 - values captured by call sensitive assertion GA0004.
Iteration 3. Once again, Autoguard ran the two programs. Assertion GA0004 revealed that the variable `sat_geodetic`, used in the computation of `sat_lat`, was defined a conflicting value in the function, `Calculate_LatLonAlt`. Autoguard subsequently traced `sat_geodetic` into the function and generated assertions for the variables used in the computation of `sat_geodetic`.

Iteration 4. A number of the assertions generated during Iteration 03 failed after Autoguard executed the two programs. The assertion graph, shown below, revealed that failed assertion GA0012, testing the incoming parameter for `pos`, was used directly or indirectly in the computation of the other erroneous variables.
Autoguard subsequently generated assertions to trace the incoming parameter, pos, from the call (in Calc) stepped into during Iteration 3.

**Iteration 5.** Assertions GA0017 and GA0019 show that the variables vel and pos contain incorrect values after the two programs were executed. The assertions tested the values of vel and pos after calling the SGP4 function in program 1 and the SGP8 function in program 2.

We chose to focus on one variable only, namely vel. Autoguard processed vel and stepped into the function SGP4 and SGP8 in programs 1 and 2 respectively. Autoguard immediately discovered that the definition for vel in program 1 depended on different variables than the definition in program 2 and could not proceed. In particular, the definition in program 1 was:

vel->z=rdotk*uz+rfdotk*vz;
and the definition in program 2 was:

\[ \text{vel} -> w = \text{tmpVel.v[3]}; \]

At this point, Autoguard proceeded to identify the closest clone(s) that contained variables used, directly or indirectly, in the computation of \( \text{vel} \). Autoguard discovered a clone that contained the variable \( \text{tsince} \) that was used in the computation of \( \text{vel} \). This clone resided in the function \( \text{Calc} \) (prior to the call to SGP4 and SGP8, in programs 1 and 2 respectively, stepped into during iteration 5). Figure 6-35 depicts programs 1 and 2, the program point at which Autoguard could not continue to trace the data flow, and the program points, discovered in the clone, from which the construction of assertions continued.

**Figure 6-34: Autoguard case study 3 – Autoguard results for iterations 6 and 7.**

*Iteration 6.* The assertion generated for the variables found within the clones, SPA0001, failed after Autoguard ran the two programs. Autoguard subsequently generated three assertions to test the variables used in the computation of the variable, \( \text{tsince} \), tested by this assertions, as shown in Figure 6-34.

*Iteration 7.* Assertion GA0022, testing the value of \( \text{jul_epoch} \) computed by the function \( \text{Julian_Date_of_Epoch} \), failed after the two programs were executed. Autoguard stepped into the function and generated six further assertions.

The source of the problem was discovered after the two programs were run. In particular, assertion GA0025 revealed that the variable \( \text{day} \) was being assigned a con-
flicting value that was ultimately responsible for producing the incorrect value for the `sat_lat` traced from iteration 1. The incorrect statement:

```c
  day = modf(epoch*1E-2, &year)*1E3;
```

required the following correction:

```c
  day = modf(epoch*1E-3, &year)*1E3;
```

Once this correction was applied, the satellites were reported in the correct positions.
<table>
<thead>
<tr>
<th>Original PREDICT version (using SGP4)</th>
<th>Ported PREDICT version (using SGP8)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SingleTrack</strong></td>
<td><strong>iteration 1</strong></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>PreCalc</td>
<td>PreCalc</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Calc</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>display sat_lat</td>
<td>display sat_lat</td>
</tr>
<tr>
<td><strong>PreCalc</strong></td>
<td><strong>iteration 2</strong></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Julian_Date_of_Epoch</td>
<td>Julian_Date_of_Epoch</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>tsince = ...</td>
<td>tsince = ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>if near earth</td>
<td>if near earth</td>
</tr>
<tr>
<td>satellite then</td>
<td>satellite then</td>
</tr>
<tr>
<td>SGP4</td>
<td>SGP8</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>SDP4</td>
<td>SDP4</td>
</tr>
<tr>
<td>end-if</td>
<td>end-if</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Calculate_LatLonAlt</td>
<td>Calculate_LatLonAlt</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>SGP4</strong></td>
<td><strong>iteration 3</strong></td>
</tr>
<tr>
<td>algorithm to compute SGP4</td>
<td>algorithm to compute SGP8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>vel-&gt;z = rdotk<em>uz+rfdotk</em>vz;</td>
<td>vel-&gt;w = tmpVel.v[3];</td>
</tr>
</tbody>
</table>

Figure 6-35: Autoguard case study 3 - Iteration summary for iterations 1, 2 and 3.
6.3.1 Case Study 3 Summary

This case study has demonstrated the use of cloning to resynchronize, and construct further assertions, when the programs differ in such a way that Autoguard can no longer match variables. In this particular case study, the origin of the error occurred before the resynchronization point, and allowed Autoguard to trace the error from the resynchronization point and isolate the cause. Seven iterations, including the resynchronization, were required to isolate the cause of the error. This case study also showed that call sensitive assertions are useful when tracing variables across function calls.

6.4 Summary

The three case studies presented in this chapter have successfully highlighted the success of Automatic Relative Debugging. In all three case studies, the source of the fault was discovered by the Automatic Relative Debugger with little human intervention. In particular, the case studies have highlighted the following contributions of our research:

- The ability to automatically construct assertions that trace an error back to the origin of the fault.
- The ability to identify variables that have been renamed and construct assertions accordingly.
- The ability to create assertions that trace an error through a number of function calls.
- The ability to resynchronize, and continue constructing assertions, when portions of the programs differ.
- The ability to construct assertions and perform automatic Relative Debugging with programs that written in different source languages.
7 Future Directions and Conclusion

Relative Debugging is a powerful paradigm for locating errors in programs that have undergone software evolution. This thesis has presented three distinct contributions that have significantly enhanced the Relative Debugging paradigm, empowering users to become more effective when debugging programs that have undergone software evolution. These contributions are:

- *A Systematic Approach to Relative Debugging.*
- *Data Flow Browsing for Relative Debugging.*
- *Automatic Relative Debugging.*

These contributions have enhanced the Relative Debugging paradigm and allow errors to be localized with little interaction required from the user. Minimizing the user’s involvement reduces the cost of debugging programs that have undergone software evolution. Automating the Relative Debugging paradigm reduces the need for users to have a detailed knowledge of the programs under consideration, thereby improving productivity, and has a significant impact on current debugging practices.

The work presented in this thesis provides a solid foundation that can isolate faults quickly and easily. This chapter presents a number of areas that could benefit from further research and development, leading to an improvement in current functionality and usability. The chapter ends with a summary of our work and concludes this thesis.

7.1 Visualization

Software clones are an effective technique that allows suitable data structures and program locations to be identified when the programs differ and matching data structures cannot be identified, as discussed in Section 5.2. If the two program versions differ sufficiently such that variables can no longer be matched, software clones are employed to match variables and generate assertions.

If the assertion placed at the end of the clone fails, the error must occur before the assertion and the automatic construction of assertions may resume. On the other hand, if the assertion succeeds, the error must originate between the created assertion
and the assertion placed at the point where the two programs began to differ. In this case, the user must uncover the pertinent facts, such as control flow, call chain etc, required to debug the problem manually. This task can become extremely tedious and error prone when the region of code between the two assertions is large. It would be advantageous if information could be automatically discovered and presented to the user.

7.2 Program Analysis

Our approach to automatic Relative Debugging cannot currently disambiguate, and match, multiple definitions that may reach a particular variable use (as discussed in Section 5.1.1.2). To eliminate the ambiguity when several definitions exist, we place the programs in Static Single Assignment (SSA) form to ensure that only one definition may ever reach the use of a variable in a particular computation. The resulting assertion, which is placed at the point where the control paths containing the multiple definitions converge in the control flow, represents the multiple definitions. Consequently, a region of code that is producing incorrect values is identified rather than individual statements. When the representative assertion fails, the user must determine which one of the multiple definitions has produced the incorrect result.

Future work could involve the investigation of more complex program analysis techniques in order to eliminate the ambiguity when there are multiple definitions for a variable. It is likely that the ambiguity cannot be eliminated in all cases, but removing some cases will yield better results for the user.

7.3 Refactoring

Refactoring, often attributed to Fowler [51], improves the design of an existing software system. The overall goal is to improve readability, structure, performance, abstraction, maintainability or other features generally desired by a well-engineered software system. Refactoring is performed by applying small, well-defined, transformations to the internal structure of the software system while maintaining the external behaviour. An evolving catalogue of transformations that are commonly applied to software systems is maintained and published by Fowler [51].
The task of refactoring can be performed manually by a developer, or it can be automated with supporting tools. A handful of tools that provide refactoring support currently exist and either perform the behaviour preserving transformation(s) or identify segments of code that are candidates for certain transformations. The refactoring tools that are currently available only support a limited number of transformations; no tool supports the entire catalogue of transformations.

Our approach for automatic Relative Debugging could be used to verify that applied transformations, made by a developer or an automatic tool, do not alter the external behaviour of the system. In the case where a transformation changes the external behaviour, an automatic Relative Debugger could be used to isolate the introduced error. Empirical research could be conducted to identify which refactorings could be identified by an automatic Relative Debugger. The results could provide the basis for further research, and steer improvements to the underlying automatic Relative Debugging foundation detailed in this thesis.

7.4 Conclusion

Programs continually evolve due to software engineering activities such as maintenance, enhancements or porting from one language or machine to another. To assist developers locate errors in programs that are subjected to activities that result in program evolution, Relative Debugging was proposed in 1994. Relative Debugging is a unique and powerful paradigm that utilizes the existing version to localize faults; faults are isolated by identify divergences that occur between key data structures in the two versions as they are simultaneously executed. With this approach, the developer is less concerned with the actual state of the modified program and more concerned with finding when and where the difference between the old and new versions occur.

Relative Debugging requires the user to identify the corresponding data structures and program points at which comparisons should be performed when the two program versions are executed concurrently. The choice of data structures and program points must be determined by the user based on some knowledge of the programs. Even with a comprehensive understanding of the two programs, this task can become
laboriously, particularly in large scientific programs that contain complex control and data flows.

This thesis has presented three distinct contributions that have significantly enhanced the Relative Debugging paradigm, empowering users to become more effective when debugging programs that have undergone software evolution. The first of these contributions is:

1. **A Systematic Approach to Relative Debugging**

   Relative Debugging has proven to be extremely useful for debugging programs that have undergone software evolution. From reports and observations, we have identified a general pattern of usage that allows errors to be quickly and effectively discovered. We presented a systematic approach to provide a clear, well defined framework that allows users, both experienced and inexperienced, to quickly and effectively locate the source of an error when using Relative Debugging. Further, the systematic approach provides an important framework that allows us to achieve our other two contributions, namely data flow browsing for Relative Debugging and Automatic Relative Debugging.

To remove the burden of identifying the corresponding data structures and program points, the second contribution addressed by our research and presented in this thesis is:

2. **Data Flow Browsing for Relative Debugging**.

   To locate variable definitions and construct useful assertions requires that the user has a detailed knowledge of the two programs under consideration. Without a detailed knowledge of the two programs, the task of locating definitions becomes difficult and time consuming. This is especially true in complex multi-file programs where there may be multiple definitions that are sparsely scattered throughout the two programs.
A Relative Debugging Data Flow Browser locates and displays the definitions that may assign a value to a selected variable in a particular computation. Such a tool provides invaluable support for Relative Debugging by reducing the complexity of identifying variable definitions. Furthermore, it decreases the need for users to have a detailed knowledge of the programs under consideration, making Relative Debugging significantly easier than in the past, and subsequently improves productivity.

The third contribution of our research that further eliminates the need for users to have a detailed knowledge of the two programs is:

3. **Automatic Relative Debugging.**

We presented an approach that automates the Relative Debugging methodology by identifying the data structures and program points for the user. In a manner very similar to the manual approach, we propose a systematic approach that constructs assertions to trace the flow of an error back to the origin of the fault. Our approach requires little guidance by the user and eliminates the need for users to have a detailed knowledge of the two programs under consideration. Such automation radically improves the current method for debugging programs that have undergone software evolution and dramatically improves productivity.

A number of existing techniques have been leveraged to achieve automatic Relative Debugging and have resulted in the following achievements:

4. An approach that allows matching data structures and program locations to be identified such that assertions can be automatically constructed.

Our approach uses the data flow properties of the two programs under consideration to automatically identify the data structures and program points where assertions should be created and comparisons performed when the two programs are executed.
5. *An approach that allows suitable data structures and program locations to be identified when the programs differ and matching data structures cannot be identified when tracing an error back through a program.*

When the data flow differs between the two programs and data structures cannot be matched, it is not possible to generate assertions automatically. In this case, we use cloning techniques as a novel approach that allow matching data structures and program points to be identified, and provide a suitable program points from which the automatic construction of assertions can resume.

6. *An approach that allows variables that have been renamed to be matched such that assertions can be automatically constructed.*

We present a novel approach that allows us to identify renamed variables and automatically construct assertions in such cases.

The implementation of an automatic Relative Debugger has demonstrated that these contributions combine to provide a powerful tool for debugging applications that have undergone software evolution. The effectiveness of these contributions was presented in the three case studies presented in Chapter 6. Each case study showed how an automatic Relative Debugger is useful when an application is subjected to software engineering tasks that result in program evolution.

These contributions have enhanced the Relative Debugging paradigm and allow errors to be localized with little human interaction. Minimizing the user’s involvement reduces the cost of debugging programs that undergone software evolution. Automating the Relative Debugging paradigm reduces the need for users to have a detailed knowledge of the programs under consideration, thereby improving productivity and has a significant impact on current debugging practices.
Appendix A: VSGuard Interfaces

API Functions

StartInterp

Description: Called by the client to start guard debugging.

HRESULT StartInterp /*[in]*/ BSTR invoke1,
       /*[in]*/ BSTR invoke2)

AddAssert

Description: Called by whoever wants to add an assertion. Return an id to allow client to keep track of assertions.

HRESULT AddAssert /*[in]*/ BSTR proc1,
       /*[in]*/ BSTR var1,
       /*[in]*/ BSTR file1,
       /*[in]*/ long line1,
       /*[in]*/ BSTR scope1,
       /*[in]*/ BSTR proc2,
       /*[in]*/ BSTR var2,
       /*[in]*/ BSTR file2,
       /*[in]*/ long line2,
       /*[in]*/ BSTR scope2,
       /*[in]*/ long apopup,
       /*[in]*/ long astop,
       /*[in]*/ double lower,
       /*[in]*/ double upper,
       /*[in]*/ long use,
       /*[in,out]*/ long * ID)
StopInterp

**Description:** Called by the client to stop guard debugging. Does not stop vs.net debugger.

`HRESULT StopInterp ()`

StopDebugger

**Description:** Called by client to stop the vs.net debugger. Calls an the automation routine Stop().

`HRESULT StopDebugger ()`

DeleteAssert

**Description:** Deletes an assertion created by AddAssert.

`HRESULT DeleteAssert (/*[in]*/ long ID)`

ClearAsserts

**Description:** Delete all assertions.

`HRESULT ClearAsserts ()`

RegisterGuardClient

**Description:** Called by whoever wants to be informed of guard events.

`HRESULT RegisterGuardClient (/*[in]*/ struct IGuardClient * gc, /*[in,out]*/ long * myref)`

UnRegisterGuardClient

**Description:** Called by whoever wants to no longer be informed of guard events.

`HRESULT UnRegisterGuardClient (/*[in]*/ long myref)`
VerifyVarWithPDB

**Description:** Checks that the specified variable exists within the specified program (using the pdb file).

```c
HRESULT VerifyVarWithPDB /*[in]*/ BSTR bVar,
   /*[in]*/ BSTR bProjectName,
   /*[in]*/ BSTR bFilename,
   /*[in]*/ BSTR bScope,
   /*[in]*/ long lLineNum,
   /*[in,out]*/ BSTR * bVarType,
   /*[in,out]*/ long * lResult,
   /*[in,out]*/ long * plLanguage,
   /*[in,out]*/ BSTR * pbLixicalParentName,
   /*[in,out]*/ BSTR * pbGlobAccessName)
```

GetAssert

**Description:** Retrieves the details for a specified assertion.

```c
HRESULT GetAssert /*[in,out]*/ BSTR * proc1,
   /*[in,out]*/ BSTR * var1,
   /*[in,out]*/ BSTR * file1,
   /*[in,out]*/ long * line1,
   /*[in,out]*/ BSTR * scope1,
   /*[in,out]*/ BSTR * proc2,
   /*[in,out]*/ BSTR * var2,
   /*[in,out]*/ BSTR * file2,
   /*[in,out]*/ long * line2,
   /*[in,out]*/ BSTR * scope2,
   /*[in,out]*/ long * apopup,
   /*[in,out]*/ long * astop,
   /*[in,out]*/ double * lower,
   /*[in,out]*/ double * upper,
   /*[in,out]*/ long * use,
   /*[in,out]*/ long * ID)
```
Callback API

DebugResult

Description: Called when an assertion has been fired i.e., a comparison has occurred.

HRESULT DebugResult(/*[in]*/ long idx,
/*[in]*/ BSTR type,
/*[in]*/ long iszero,
/*[in]*/ BSTR title,
/*[in]*/ BSTR data,
/*[in]*/ BSTR name1,
/*[in]*/ BSTR data1,
/*[in]*/ BSTR name2,
/*[in]*/ BSTR data2)

HalfResult

Description: Essentially the same as DebugResult, except this is called when we have just retrieved a value (hit one half of an assertion) but we haven’t performed the comparison yet. Allows us to see how many times each side of the comparison has been it.

HRESULT HalfResult (/*[in]*/ long idx,
/*[in]*/ long halfref,
/*[in]*/ BSTR title,
/*[in]*/ BSTR name1,
/*[in]*/ BSTR data1 )

StatusMsg

Description: Status messages sent from VSGuard.

HRESULT StatusMsg(BSTR msg)
ErrorMsg

Description: Error message sent from VSGuard.

HRESULT ErrorMsg(BSTR msg)

DebugMsg

Description: Debug messages sent from VSGuard.

HRESULT DebugMsg(BSTR msg)

GuardEvent

Description: Sent by VSGuard when a VSGuard occurs.

HRESULT GuardEvent(GuardEventEnum evnt, BSTR msg)
Appendix B: Source Code Listings

Autoguard Case Study 2 - Original Shallow Water Fortran Source Code

program shallow

"Finite difference model of shallow water equations based on
""The dynamics of finite-difference models of the shallow-water equations"
by R. Sadourny, JAS, 32, 1975.
Code from "An introduction to three-dimensional climate modelling"
by Washington and Parkinson.

parameter(m=100, n=100, imax=m+1, jmax=n+1)
parameter(a=1.e6, dt=90., dx=1.e5, dy=1.e5, alpha=0.001)
parameter(itmax=10, mprint=1)

common /u/ u(imax, jmax)
common /v/ v(imax, jmax)
common /p/ p(imax, jmax)
common /unew/ unew(imax, jmax)
common /vnew/ vnew(imax, jmax)
common /pnew/ pnew(imax, jmax)
common /uold/ uold(imax, jmax)
common /vold/ vold(imax, jmax)
common /pold/ pold(imax, jmax)
common /cu/ cu(imax, jmax)
common /cv/ cv(imax, jmax)
common /z/ z(imax, jmax)
common /h/ h(imax, jmax)
common /psi/ psi(imax, jmax)
common /ketot/ ketot
common /ncycle/ ncycle
common /enstot/ enstot

real ketot

tdt=dt
time=0.
pi=4.*atan(1.)
tpi=pi+pi
di=tpi/float(m)
dj=tpi/float(n)

*** initial values of the streamfunction
do 50 j=1, jmax
   do 50 i=1, imax
      psi(i, j) = a*sin((float(i)-0.5)*di)*sin((float(j)-0.5)*dj)
   50 continue

*** initialize velocities
do 60 j=1, n
   do 60 i=1, m
      u(i+1, j) = -(psi(i+1, j+1) - psi(i+1, j))/dy
      v(i, j+1) = -(psi(i+1, j+1) - psi(i, j+1))/dx
   60 continue

*** periodic continuation
do 70 j=1, n
   do 70 i=1, m
      u(i+1, n+1) = u(i+1, 1)
      v(i, 1) = v(i, n+1)
   70 continue

u(1, n+1) = u(m+1, 1)
v(m+1, 1) = v(1, n+1)
do 86 j=1, imax
do 86 i=1, jmax
uold(i, j) = u(i, j)
vold(i, j) = v(i, j)
pold(i,j) = 50000.
p(i,j) = 50000.
86 continue
c *** print initial values
c write(*,390) n,m,dx,dy,dt,alpha
390 format(' Number of points in the X direction',i8,/, 1
' Number of points in the Y direction',i8,/, 2
' Grid spacing in the X direction  ',f8.0,/, 3
' Grid spacing in the Y direction  ',f8.0,/, 4
' Time step                          ',f8.3,/, 5
'min = min0(m,n)
c write(*,391) (pold(i,i),i=1,mnmin)
391 format(/,' Initial diagonal elements of P ',/,/(8e16.6))
c write(*,392) (vold(i,i),i=1,mnmin)
392 format(/,' Initial diagonal elements of U ',/,/(8e16.6))
c write(*,393) (vold(i,i),i=1,mnmin)
393 format(/,' Initial diagonal elements of V ',/,/(8e16.6))
cycle = 0
90 ncycle = ncycle +1
CASSERT test u[1..100][1..100]
95 continue
c write(*,909) ncycle
909 format(/,' Ncycle ',/,/(i6)
c *** compute U, V, z and h
fsdx = 4./dx
fsdy = 4./dy
ptot=0.
pmean=0.
knetot=0.
etot=0.
enstot = 0.
do = 1,n
 do i=1,m
 pmean = pmean+p(i,j)
 end do
 end do
 pmean = pmean/(m*n)
do 100 j=1,n
 do 100 i=1,m
 cu(i+1,j) = 0.5*p(i+1,j)+p(i,j)*u(i+1,j)
cv(i,j+1) = 0.5*p(i,j+1)+p(i,j)*v(i,j+1)
z(i+1,j+1) = (fsdx*(v(i+1,j+1)-v(i,j+1)))+fsdy*(u(i+1,j+1)
1 -u(i+1,j)))/(p(i,j)+p(i+1,j)+p(i+1,j+1)+p(i,j+1))
h(i,j) = p(i,j)*0.25*(u(i+1,j)+u(i,j)+u(i,j+1)
1 +v(i,j+1)+v(i,j))
knetot = ketot + p(i,j)*0.25*(u(i+1,j)+u(i,j)+u(i,j+1)
1 +v(i,j+1)+v(i,j))
ptot = ptot+(p(i,j)-pmean)**2
 etot = etot+h(i,j)
enstot = enstot + z(i+1,j+1)**2 * 0.25*
1 (p(i,j)+p(i+1,j)+p(i+1,j)+p(i,j+1))
100 continue
ptot = 0.5*ptot/(m*n)
knetot = ketot/(m*n)
etot = etot/(m*n)
enstot = enstot/(m*n)
c *** periodic continuation
do 110 j=1,n
cu(1,j) = cu(m+1,j)
cv(m+1,j+1) = cv(1,j+1)
z(1,j+1) = z(m+1,j+1)
h(m+1,j) = h(1,j)
110 continue
do 115 i=1,m
 cu(i+1,n+1) = cu(i+1,1)
cv(1,1) = cv(i,n+1)
z(i+1,1) = z(i+1,n+1)
h(i,n+1) = h(1,1)
115 continue
 cu(1,n+1) = cu(m+1,1)
cv(m+1,1) = cv(1,n+1)
z(1,1) = z(m+1,n+1)
h(m+1,n+1) = h(1,1)
c *** compute new values of u, v and p
tdts8 = tdt/8.
tdtsdx = tdt/dx
\[ tdt_{\text{dy}} = \frac{tdt}{dy} \]

**C*** ENERGY CONSERVING

\[
\begin{align*}
\text{new}(i+1,j) &= \text{old}(i+1,j) + \\
1 & \quad tdt_{\text{dy}} \left( z(i+1,j+1)(cv(i+1,j+1)+cv(i,j+1)) + \\
2 & \quad z(i+1,j)(cv(i+1,j)+cv(i,j)) \right) + \\
3 & \quad -tdt_{\text{dx}}(h(i+1,j)-h(i,j)) \\
\text{new}(i+1,j) &= \text{old}(i+1,j) - \\
1 & \quad tdt_{\text{dx}} \left( z(i+1,j+1)(cu(i+1,j+1)+cu(i+1,j)) + \\
2 & \quad z(i,j+1)(cu(i,j+1)+cu(i,j)) \right) \\
3 & \quad -tdt_{\text{dy}}(h(i,j+1)-h(i,j))
\end{align*}
\]

**C*** POTENTIAL ENSTROPHY CONSERVING

\[
\begin{align*}
\text{new}(i+1,j) &= \text{old}(i+1,j) + \\
1 & \quad tdt_{\text{dy}}(z(i+1,j+1)+z(i+1,j))(cv(i+1,j+1)+cv(i,j+1)+cv(i,j)) + \\
2 & \quad +cv(i+1,j)tdt_{\text{dx}}(h(i+1,j)-h(i,j)) \\
\text{new}(i+1,j) &= \text{old}(i+1,j) - tdt_{\text{dx}}(z(i+1,j+1)+z(i,j+1)) \\
1 & \quad (cu(i+1,j+1)+cu(i,j+1)+cu(i,j)+cu(i+1,j)) + \\
2 & \quad -tdt_{\text{dy}}(h(i,j+1)-h(i,j)) \\
\text{new}(i,j) &= \text{old}(i,j) - tdt_{\text{dx}}(cu(i+1,j)-cu(i,j)) + \\
1 & \quad -tdt_{\text{dy}}(cv(i+1,j)-cv(i,j))
\end{align*}
\]

200 continue

**C*** periodic continuation

do 210 j=1,n
\text{new}(1,j) = \text{old}(1,j)
\text{new}(m+1,j) = \text{old}(1,j)
\text{new}(1,j+1) = \text{old}(i,j+1)
\text{new}(i,j+1) = \text{old}(i,j+1)
210 continue

**C*** periodic continuation

do 215 i=1,m
\text{new}(i,1+n) = \text{old}(i,1+n)
\text{new}(i,1) = \text{old}(i,1)
\text{new}(i+1,n) = \text{old}(i+1,n)
\text{new}(i+1,n) = \text{old}(i+1,n)
215 continue

if(ncycle.gt.itmax) write(*,220)
220 format(’ ***** End of Shallow ***** ’)

if(ncycle.gt.itmax) stop

time = time+dt
if(mod(ncycle,mprint).ne.0) go to 370
ptime = time/86400.
write(*,350) ncycle,ptime,ptot,ketot,ptot+ketot,ensot

**C*** diagonal elements of P

write(*,355) (pnew(i,j),i=1,mmin)
355 format(/,’ Diagonal elements of P ’,/,(/,e16.6)

**C*** diagonal elements of U

write(*,360) (unew(i,j),i=1,mmin)
360 format(/,’ Diagonal elements of U ’,/,(/,e16.6)

**C*** diagonal elements of V

write(*,365) (vnew(i,j),i=1,mmin)
365 format(/,’ Diagonal elements of V ’,/,(/,e16.6)

370 if(ncycle.le.1) go to 310

do 300 j=1,n
do 300 i=1,m
old(i,j) = u(i,j)+alpha*(new(i,j)-2.*u(i,j)+old(i,j))
void(i,j) = v(i,j)+alpha*(new(i,j)-2.*v(i,j)+void(i,j))
pold(i,j) = p(i,j)+alpha*(pnew(i,j)-2.*p(i,j)+pold(i,j))
old(i,j) = new(i,j)
new(i,j) = new(i,j)
p(i,j) = pnew(i,j)
300 continue

**C*** periodic continuation

do 320 j=1,n
old(m+1,j) = old(1,j)
void(m+1,j) = void(1,j)
pold(m+1,j) = pold(1,j)
new(m+1,j) = new(1,j)
320 continue

do 325 i=1,m
old(i,n+1) = old(i,1)
195
vold(i,n+1) = vold(i,1)
pold(i,n+1) = pold(i,1)
u(i,n+1) = u(i,1)
v(i,n+1) = v(i,1)
p(i,n+1) = p(i,1)

325 continue
vold(m+1,n+1) = vold(1,1)
pold(m+1,n+1) = pold(1,1)
u(m+1,n+1) = u(1,1)
v(m+1,n+1) = v(1,1)
p(m+1,n+1) = p(1,1)
go to 90

310 tdt = tdt+tdt
    do 400 j=1,jmax
    do 400 i=1,imax
        uold(i,j) = u(i,j)
vold(i,j) = v(i,j)
pold(i,j) = p(i,j)
u(i,j) = unew(i,j)
v(i,j) = vnew(i,j)
p(i,j) = pnew(i,j)
        400 continue
    go to 90
end
using System;

namespace ShallowWaterCSharp1
{
    class Class1
    {
        //static int m = 32; /* Number of points in x direction */
        //static int n = 32; /* Number of points in y direction */
        static int m = 4; /* Number of points in x direction */
        static int n = 4; /* Number of points in y direction */
        static int imax = m + 1;
        static int jmax = n + 1;

        static double a = 1.0e6; /* Nominally the radius of the earth but here just length scale*/
        static double dt = 90.0; /* Time step in seconds */
        static double dx = 1.0e5; /* Grid spacing in x direction */
        static double dy = 1.0e5; /* Grid spacing in y direction */
        static double alpha = 0.001; /* Asselin time filter parameter */

        //static int itmax = 950; /* Number of time steps in run */
        //static int mprint = 50; /* Print diagnostics every mprint steps */
        static int itmax = 5; /* Number of time steps in run */
        static int mprint = 5; /* Print diagnostics every mprint steps */

        static void skip()
        {
        }
    }

    // initialise
    static void initialise(ref float[,] u, ref float[,] v, ref float[,] p, ref float[,] psi, ref float[,] uold, ref float[,] vold, ref float[,] pold, ref float di, ref float dj)
    {
        int i, j;
        skip(); // blank line for incoming param
        skip();

        /* *** initialise values of the streamfunction */
        for (j = 0; j < jmax; j++)
        {
            for (i = 0; i < imax; i++)
            {
                psi[i, j] = (float) a * (float) (Math.Sin((double)((i+0.5)*di)) * Math.Sin((double)((j+0.5)*dj)));
                skip();
            }
        }

        Console.WriteLine();

        /* *** initialise velocities */
        for (j = 0; j < n; j++)
        {
            for (i = 0; i < m; i++)
            {
                /* initialise velocities */
                skip();
            }
        }
    }
}
for (j = 0; j < n; j++)
    for (i = 0; i < m; i++)
        u[0, j] = u[m, j];
        v[m, j + 1] = v[0, j + 1];
        skip();
    for (i = 0; i < m; i++)
        u[i + 1, n] = u[i + 1, 0];
        v[i, 0] = v[i, n];
        skip();
    u[0, n] = u[m, 0];
    v[m, 0] = v[0, n];

for (j = 0; j < jmax; j++)
    for (i = 0; i < imax; i++)
        uold[i, j] = u[i, j];
        vold[i, j] = v[i, j];
        pold[i, j] = 50000;  // Free surface height * gravitational accel'nn */
        p[i, j] = 50000;
        skip();
}

Console.WriteLine("Number of points in the X direction   {0:D}", n);
Console.WriteLine("Number of points in the Y direction   {0:D}", m);
Console.WriteLine("Grid spacing in the X direction       {0:F}", dx);
Console.WriteLine("Grid spacing in the Y direction       {0:F}", dy);
Console.WriteLine("Time step                             {0:F}", dt);
Console.WriteLine("Time filter parameter                 {0:F}", alpha);

// diag: Calculate global integrals of kinetic and potential
// energy and potential enstrophy
static void diag(int ncycle,
                 float fsdx,
                 float fsdy,
                 ref float time,
                 ref float[,] p,
                 ref float[,] u,
                 ref float[,] v,
                 ref float[,] h,
                 ref float[,] z,
                 ref float[,] cu,
                 ref float[,] cv,
                 out float PTOT,
                 out float ketot,
                 out float etot,
                 out float enstot,
out float pmean)
{
    int i,j;
    /* *** compute U, V, z and h */
    skip();    // Blank line to place incoming param assert on.
    skip();
    PTOT=0;
    pmean = 0;
    ketot=0;
    etot=0;
    enstot = 0;
    skip();
    for (j = 0;
        j < n;
        j++)
    {    
        for (i = 0;
            i < m;
            i++)
        {
            pmean = pmean+p[i,j];
            skip();
        }
    pmean = pmean/(m*n);
    skip();
    for (j = 0;
        j < n;
        j++)
    {    
        for (i = 0;
            i < m;
            i++)
        {
            cu[i+1,j] = (float) (0.5*(p[i+1,j]+p[i,j])*u[i+1,j]);
            cv[i,j+1] = (float) (0.5*(p[i,j+1]+p[i,j])*v[i,j+1]);
            z[i+1,j+1] = (fsdx*(v[i+1,j+1]-v[i,j+1])-fsdy*(u[i+1,j+1]-
                          u[i,j]))/(p[i,j]+p[i+1,j]+p[i+1,j+1]+p[i,j+1]);
            h[i,j] = (float)
                      (p[i,j]*0.25*(u[i+1,j]*u[i+1,j]+u[i,j]*u[i,j]+v[i,j+1]*v[i,j+1]+v[i,j]*v[i,j]));
            ketot = (float) (ketot +
                       p[i,j]*0.25*(u[i+1,j]*u[i+1,j]+u[i,j]*u[i,j]+v[i,j+1]*v[i,j+1]+v[i,j]*v[i,j]));
            PTOT = PTOT * (p[i,j]-pmean) * (p[i,j]-pmean);  // *** BUG 01
            PTOT = PTOT + ( p[i,j]-pmean ) * ( p[i,j]-pmean ) ;
            etot = etot+h[i,j];
            enstot = (float) (enstot + (z[i+1,j+1]*z[i+1,j+1]) *
                           0.25*(p[i+1,j]+p[i+1,j]+p[i+1,j]+p[i,j]+p[i,j]));
            skip();
        }
    }
    PTOT *= (float) (0.5/(m*n));
    ketot /= (m*n);
    etot /= (m*n);
    enstot /= (m*n);
    skip();
    /*
      if (ncycle <= itmax)
      {
        time = time+dt;
        if (ncycle % mprint == 0)
        {
          ptime = time/86400;
          Console.WriteLine("""--- "");
          Console.WriteLine("""Cycle number [0:D5]", ncycle);
          Console.WriteLine("""Model time in days [0:F]", ptime);
          Console.WriteLine("""Potential energy [0:F]", PTOT);
          Console.WriteLine("""Kinetic energy [0:F]", ketot);
        }
      }
    */
static void calcuvzh(float tdt,
    ref float[,] p,
    ref float[,] u,
    ref float[,] v,
    ref float[,] cu,
    ref float[,] cv,
    ref float[,] z,
    ref float[,] h,
    ref float[,] uold,
    ref float[,] void,
    ref float[,] polold,
    ref float[,] unew,
    ref float[,] vnew,
    ref float[,] pnew)
{
    int i, j;
    float tdts8, tdtsdx, tdtsdy;

    skip();    // Testing incoming param theory.
    skip();

    //Console.WriteLine(tdt);
    //skip();

    /* *** periodic continuation */
    for (j = 0;
        j < n;
        j++)
    {
        cu[0,j] = cu[m,j];
        cv[m,j+1] = cv[0,j+1];
        z[0,j+1] = z[m,j+1];
        h[m,j] = h[0,j];
        skip();
    }

    for (i = 0;
        i < m;
        i++)
    {
        cu[i+1,n] = cu[i+1,0];
        cv[i,0] = cv[i,n];
        z[i+1,0] = z[i+1,n];
        h[i,n] = h[i,0];
        skip();
    }

    cu[0,n] = cu[m,0];
    cv[m,0] = cv[0,n];
    z[0,0] = z[m,0];
    h[m,n] = h[0,0];
    skip();

    tdts8 = (float) (tdt / 8.0);
    tdtsdx = (float) (tdt / dx);
    tdtsdy = (float) (tdt / dy);

    for (j = 0;
        j < n;
        j++)
    {
        for (i = 0;
            i < m;
            i++)
    
        /* *** ENERGY CONSERVING */
        unew[i+1,j] = uold[i+1,j]+tdts8*
            z[i+1,j+1]*((cv[i+1,j+1]+cv[i,j+1])+z[i+1,j]*(cv[i+1,j]+cv[i,j]))-tdtsdx*h[i+1,j]-
h[i,j]);
  vnew[i,j+1] = void[i,j+1]-
  tdst8*(z[i+1,j+1]*(cu[i+1,j+1]+cu[i+1,j])+z[i,j+1]*(cu[i,j+1]+cu[i,j]))-
  tdstdy*(h[i,j+1]-h[i,j]);
  pnew[i,j] = pold[i,j]-tdtsdx*(cu[i+1,j]-cu[i,j])-tdtsdy*(cv[i,j+1]-
  cv[i,j]);
  skip();
}

/** *** periodic continuation */
for (j = 0; j < n; j++)
{
  unew[0,j] = unew[m,j];
  vnew[m,j+1] = vnew[0,j+1];
  pnew[m,j] = pnew[0,j];
  skip();
}

for (i = 0; i < m; i++)
{
  unew[i+1,n] = unew[i+1,0];
  vnew[i,0] = vnew[i,n];
  pnew[i,n] = pnew[i,0];
  skip();
}

unew[0,n] = unew[m,0];
  vnew[m,0] = vnew[0,n];
  pnew[m,n] = pnew[0,0];
  skip();
}

#if ddd
  static void timetend(ref float[,] dudt,
                      ref float[,] dvdt,
                      ref float[,] dpdt,
                      ref float[,] z,
                      ref float[,] cv,
                      ref float[,] cu,
                      ref float[,] h)
  {
    int i,j,ip,jp;
    float invdxTmp, invdyTmp;
    invdxTmp = (float) (1/dx);
    invdyTmp = (float) (1/dy);
    skip();
    for(j = jstart; j <= jend; j++)
    {
      jp = (j+1) % n;
      skip();
      for (i = 0; i < m; i++)
      {
        ip = (i+1) % m;
        skip();
        /** *** ENERGY CONSERVING */
        dudt[ip,j] = (float) (0.125 * ( z[ip,jp]*(cv[ip,jp]+cv[i,j]) +
                                  z[ip,j]*(cv[ip,j]+cv[i,j]) ) - (h[ip,j]-h[i,j])*invdxTmp);
        dvdt[i,jp] = (float) (-0.125 * ( z[ip,jp]*(cu[ip,jp]+cu[i,j]) +
                                  z[i,jp]*(cu[i,j]+cu[i,j]) ) - (h[i,jp]-h[i,j])*invdyTmp);
        dpdt[i,j] = (float) (-0.125 * ( z[i,jp]*(cv[i,j]+cv[i,j]) +
                                  z[i,j]*(cv[i,jp]+cv[i,j]) ) - (h[i,j]-h[i,j])*invdyTmp);    
      }
    }
  }
#endif
static void tstep(int ncycle,
    ref float[,] uold,
    ref float[,] vold,
    ref float[,] pold,
    ref float[,] u,
    ref float[,] v,
    ref float[,] p,
    ref float[,] unew,
    ref float[,] vnew,
    ref float[,] pnew,
    ref float[,] dudt,
    ref float[,] dvdt,
    ref float[,] dpdt,
    ref float tdt)
{
    int i2,j2;
    int i3,j3;
    int i4,j4;
    skip();    // blank line for incoming param
    skip();

    if (ncycle > 1)
    {
        /* Don't apply time filter on first step */
        //for (j2 =  n;      // *** BUG 02
        //  j2 <= n;
        //  j2++)
        for (j2 =  0;
            j2 < n;
            j2++)
        {
            for (i2 = 0;
                i2 < m;
                i2++)
            {
                uold[i2,j2] = (float) (u[i2,j2]+alpha*(unew[i2,j2]-2.0 *
                    u[i2,j2]+uold[i2,j2]));
                vold[i2,j2] = (float) (v[i2,j2]+alpha*(vnew[i2,j2]-2.0 *
                    v[i2,j2]+vold[i2,j2]));
                pold[i2,j2] = (float) (p[i2,j2]+alpha*(pnew[i2,j2]-2.0 *
                    p[i2,j2]+pold[i2,j2]));
                u[i2,j2] = unew[i2,j2];
                v[i2,j2] = vnew[i2,j2];
                p[i2,j2] = pnew[i2,j2];
                skip();
            }
        }
        /* *** periodic continuation */
        for (j4 = 0;
            j4 < n;
            j4++)
        {
            uold[m,j4] = uold[0,j4];
            void[m,j4] = void[0,j4];
            xold[m,j4] = xold[0,j4];
            u[m,j4] = u[0,j4];
            v[m,j4] = v[0,j4];
            p[m,j4] = p[0,j4];
            skip();
        }

        for (i4 = 0;
            i4 < m;
            i4++)
        {
            uold[i4,n] = uold[i4,0];
            void[i4,n] = void[i4,0];
            xold[i4,n] = xold[i4,0];
            u[i4,n] = u[i4,0];
            v[i4,n] = v[i4,0];
            p[i4,n] = p[i4,0];
        }
    }
v[i4,n] = v[i4,0];
p[i4,n] = p[i4,0];
skip();}

uold[m,n] = uold[0,0];
vold[m,n] = vold[0,0];
pold[m,n] = pold[0,0];
u[m,n] = u[0,0];
v[m,n] = v[0,0];
p[m,n] = p[0,0];
skip();
}
else
{
    tdt = tdt+tdt;
skip();
    for (j3 = 0;
        j3 < jmax;
        j3++)
    {
        for (i3 = 0;
            i3 < imax;
            i3++)
        {
            uold[i3,j3] = u[i3,j3];
vold[i3,j3] = v[i3,j3];
pold[i3,j3] = p[i3,j3];
u[i3,j3] = unew[i3,j3];
v[i3,j3] = vnew[i3,j3];
p[i3,j3] = pnew[i3,j3];
skip();
        }
    }
}

static void Main(string[] args)
{
    int ncycle;
    float tdt;
    float time;
    float pi;
    float tpi;
    float di;
    float dj;
    float fsdx;
    float fsdy;

    // Converted from global.
    float[,] u = new float[imax,jmax]; /* Zonal wind */
    float[,] v = new float[imax,jmax]; /* Meridional wind */
    float[,] p = new float[imax,jmax]; /* Pressure (or free surface height) */
    float[,] unew = new float[imax,jmax];
    float[,] vnew = new float[imax,jmax];
    float[,] pnew = new float[imax,jmax];
    float[,] psi = new float[imax,jmax]; /* Velocity streamfunction */
    float[,] dudt = new float[imax,jmax]; /* Time tendency of u */
    float[,] dvdt = new float[imax,jmax];
    float[,] dpdt = new float[imax,jmax];
    float[,] uold = new float[imax,jmax];
    float[,] vold = new float[imax,jmax];
    float[,] pold = new float[imax,jmax];
    float[,] cu = new float[imax,jmax]; /* Mass weighted u */
    float[,] cv = new float[imax,jmax]; /* Mass weighted v */
    float[,] z = new float[imax,jmax]; /* Potential enstrophy */
    float[,] h = new float[imax,jmax];
    float PTOT, ketot, etot, enstot, ptime, pmean;

    Console.WriteLine("Shallow Water Simulation (C#)");
    ncycle = 0;
skip();
tdt = (float) dt;
time = 0;
pi = 4*(float)Math.Atan((double)1);
tpi = pi + pi;
di = tpi / (float) m;
dj = tpi / (float) n;
skip();

 initialise(ref u, ref v, ref p, ref psi, ref uold, ref vold, ref pold, ref di, ref dj);
skip();

 for (ncycle = ncycle + 1;
     ncycle <= itmax;
     ncycle++)
 {
     fsdx = (float) (4 / dx);
     fsdy = (float) (4 / dy);

     diag(ncycle, fsdx, fsdy, ref time, ref p, ref u, ref v, ref h, ref z,
          ref cv, out PTOT, out ketot, out etot, out enstot, out pmean);
skip();
    /* Do the block of latitudes from jstart to jend inclusive*/
calcvzh(tdt, ref p, ref u, ref v, ref cu, ref cv, ref z, ref h, ref uold,
       ref vold, ref pold, ref unew, ref vnew, ref pnew);
skip();
    /* Calculate time tendencies of u, v and p */
    // timetend(ref dudt, ref dvdt, ref dpdt, ref z, ref cv, ref cu, ref h);
    // skip();

     time = (float) (time+dt);
skip();

     if(nycle%mprint==0)
     {
       ptime = time/86400;
skip();

       Console.WriteLine("---------------------------"SCRIPTIONS);
       Console.WriteLine("Cycle number {0:D5}", ncycle);
       Console.WriteLine("Model time in days {0:F}", ptime);
       Console.WriteLine("Potential energy {0:F}", PTOT);
       Console.WriteLine("Kinetic energy {0:F}", ketot);
       Console.WriteLine("Total energy {0:f}", PTOT+ketot);
       Console.WriteLine("Pot. Enstrophy {0:f}", enstot);
skip();
    }

     tstep(ncycle, ref uold, ref vold, ref pold, ref u, ref v, ref p, ref unew,
           ref vnew, ref pnew, ref dudt, ref dvdt, ref dpdt, ref tdt);
skip();
}

 Console.WriteLine(" ***** End of Shallow *****");
 Console.WriteLine("Press any key to continue ...");
 Console.ReadLine();
}
References


69. IBM, *IBM VisualAge for Cobol*.


137. Yang, W., S. Horwitz, and T. Reps, *Detecting Program Components With Equivalent Behaviors*. 1989, University of Wisconsin, Madison, WI, USA.


