The Development of an Incremental Debugging System

A thesis presented in partial fulfilment
of the requirements for the degree of
Master of Science
in Computer Science at
Massey University

Malcolm John M^{c}Donald

1978

Abstract

Debugging is a major area of software development that has received little attention. This thesis starts by looking at work done in the area of bug prevention, bug detection, bug location and bug correction.

A debugging system, BIAS, is proposed to help in detecting, locating and correcting bugs. Three major design goals are established. Firstly, the system should be simple and easy to understand as this will encourage use. Secondly, the system should be general so that it will be available to a large number of users. Finally, it should be incremental as this will save users' time. An incremental language, STILL, is designed to show how BIAS applies to structured languages.

The construction of the system is shown. Each data structure, and how it is used, is described. BIAS uses an interpretive system and runs threaded code on a pseudo-machine. How the threads are interpreted and how they are set up is shown next.

The use of BIAS is shown by following through an example session with the system. This consists of entering a program, editing it, and running it. As bugs show themselves, various debugging commands are used to locate the bugs. The program is then edited, and the corrections linked into the code so that it will run correctly. This cycle is repeated until no bugs remain, without at any time recompiling the whole program.

It turns out that the best way of achieving the design goals is to extend an incremental compiler host to include debugging commands. This gives a clear emphasis to the power of incremental compilers.

Acknowledgements

I would like to thank everyone at the Massey University Computer Centre and all those other people who have listened to my ideas, so often with enthusiasm. Just explaining the system has greatly aided me in the development of the system described herein. In particular, I would like to thank my supervisor, Ted Drawneek, for always having a co-operative ear, and for all the guidance and constructive criticism that was of such great value. Thanks also to Margaret Dench for her excellent job of typing when time was short, and to Phill Jenkins for general discussions about programming, which are always useful, and for the artwork on the figures.

Table of Contents

List of Figures

Chapter 1

DEBUGGING - AN INTRODUCTION

"Bloody instructions which, being learned,
Return to plague the inventor."

Macbeth in Macbeth I (vii)

When Macbeth was plotting the murder of Duncan, he realised that his plans would eventually turn against him. He decided not to go ahead, but, being easily led, he did the deed and his original thoughts were proved correct. What he needed was a good debugging system so that he could correct his mistakes before they became fatal.

Debugging has been around since Whirlwind I [Schw 71, VanT 74], yet it is one of the most neglected areas in software development [Bern 68, Gris 70, Pier 74]. This is certainly due in part to 'debugging' being a dirty word. No-one likes to admit that they make mistakes, so when the time comes to correct them, people tend to hide the fact. Consequently, each programmer thinks that he is the only one who takes such a long time to do the job, and that there is little general need for debugging aids.

What is debugging? Testing and debugging are often confused with each other as they usually overlap. When a program compiles correctly, the programmer enters data in a testing phase in which errors are detected. The programmer then tries to locate and correct these errors in a debugging phase, and the cycle is repeated. As time passes, the testing phases get longer until there appears to be no bugs (although this is often not the case). Of course, the debugging phases generally do not get shorter, and may well get longer as the errors become more obscure.

So here we arrive at a major point. Debugging takes more time than any other aspect of programming. Estimates vary from 30% to 90%

[Gain 69, Goul 75, VanT 74], so it is clear that to improve software production time, debugging is a good, if not the best, area to attack. This thesis will show the development of BIAS (Batch and InterActive System), a debugging system which collates, clarifies and simplifies existing systems.

1.1 Bug Prevention

Prevention is better than cure is a proverb well suited to debugging. While it is unlikely that all bugs can be prevented, any technique that can reduce their number or their complexity is welcome.

Every program should be well designed. This is best achieved using a top-down technique such as step-wise refinement [Wirt 71]. The modules produced should be of limited size [Your 75] and be able to stand on their own as far as possible. Interfacing is thus kept to a minimum which not only reduces the chance of having bugs, but also reduces the scope of any that do appear. This is known as bulkheading [VanT 74]. Debugging is made easier as bugs are isolated and much less likely to interact.

Style is a mark of individuality that pretends to excuse many faults, but like any writer, the programmer must use style as a beacon not a smokescreen. Good style not only reduces the number of bugs but also makes debugging much easier. One major technique is the selection of identifiers. This is the most important principle in program readability [VanT 74], although comments saying why something is done rather than what it is doing are still essential. Structuring and indentation are also valuable aids [Dora 72]. Ultimately, whatever features of style are used, they must be used consistently.

Compile-time errors are much easier to prevent than run-time errors. The prevention of compile-time errors can be done with interactive systems. There are two methods currently in use. Incremental compilers get the user to correct his syntax line by line [Ryan 66]. Interactive text editors such as EMILY [Hans 71] and GENISYS [Barr 75] actually prevent the user from making errors. EMILY works from any BNF grammar and so prevents only syntax errors, GENISYS and the system described by Lasker [Lask 74] perform static semantic checks as well.

These text editors all work by building a parse tree from the BNF grammar. All possible productions for each non-terminal are displayed and the programmers selects which production he wants by sending its associated number [Barr 75] or by pointing a light-pen to it [Hans 71, Lask 74]. The syntax of a program must consequently be correct. Unfortunately programs require a long time to enter by this method. Entry of identifiers (all the systems mentioned) and expressions (GENISYS) by typing them directly in does help. This heads the idea back towards incremental compilation.

1.2 Bug Detection

Bug detection is finding out if there are bugs. The usual tool for this is testing. As the number of possible data sets is usually astronomical, exhaustive testing is impractical, but one can improve reliability and shorten production time by using carefully selected test-cases [Buxt 69, VanT 74]. Testing should first show that the general case works. Extreme data, exploring the fringes of what is acceptable, are then tried to make sure no overflows occur. Finally, exception conditions for data that is blatantly or marginally wrong are tested to make sure errors are reported. Each type of data (general, extreme and exception) will cause its own type of error which should help to pin-point bugs. Whatever type of data is used, it must be easy to predict what output will result. If not, it will be hard to locate the bug, which may even be in the prediction of the output.

Modular test-beds allow modules to be tested individually. All globals are set by some device, and all calls from the module will be dummies. With an interactive system, the programmer can perform the action of dummy subroutines himself. He can also change parameters while the program is running and probe the boundary conditions more effectively. Bate describes such a system which resembles a breakpoint debugging system [Bate 74]. With this system, variables can be examined and altered, and breakpoints set to give the tester control anywhere in the program. A design goal for BIAS was to facilitate module testing (see section 2.5).

Two novel methods of finding bugs in systems software have been

developed by Rain [Rain 73]. The 'Bug Farm' randomly alters correct data, often providing unthought of combinations. The 'Bug Contest' offers an incentive to users to find bugs. This leaves the programmer free to repair bugs without having to spend hime time on testing. It also has a useful side-effect in that user reluctance to try the new software is overcome, and the new system is used to its fullest extent.

Unfortunately, testing shows the presence, not the absence of bugs. A formal proof of correctness, however, can show that a program is error-free, and if it is not, it can help pin-point the error(s). A bug in the F-level PL/I compiler was found by such a proof where testing had failed to detect it [Buxt 69]. Formals proofs can eliminate testing and simplify debugging while ensuring correctness. Because of this, correctness is an important area of study and is receiving much attention [Elsp 72, Lond 70].

Lowny commented that "ANY significant advance in the programming art is sure to involve very extensive automated analyses of program." [Buxt 69]. The verifying compiler described by King is an example of this [King 71]. Predicates are submitted to the compiler with the program, and the compiler does the proof. If the program is not correct, the likely source of error is pointed out. The main drawback is that as with all predicate proofs, the predicates are difficult to formulate. If the program is written with the proof in mind, the predicates will be easier to produce [Dijk 69], but even then the proof may well explode with program size.

There are compromises between formal proofs and the classical methods of debugging and testing. Less than rigorous formal proofs, amounting to a kind of disciplined desk-checking will often yield many bugs [Schw 71]. Stepwise refinement is a very informal method of proof that tends towards prevention rather than detection [Wirt 71]. Proving that critical parts of a program are correct will prevent many bugs without excessive overhead. The same will be true for proving that certain anomalies are not present in a program. Such a system is DAVE [Oste 76]. DAVE detects two types of anomalies: reference before assignment and assignment followed by no reference. This picks up uninitialised variable errors and also helps to find spelling mistakes.

All the methods of proof mentioned so far do not execute the program. However, predicates can be used at run-time to check for data anomalies. 'ON' statements in PL/I and Burroughs Extended Algol cause only low-level faults to be trapped, but are still very useful in detecting errors. Algol W 'asserts' are at a much higher level and can be as sophisticated as formal predicates [Satt 72].

1.3 Bug Location

Once a bug is known to exist, the next step is to find out exactly where and what it is. The commonest method of locating bugs is to pore over a program listing, doing a mental desk-check with the data that made the program go wrong. This is particularly true of small routines which the debugger did not write [Goul 75]. However, with larger programs, some bugs can be very hard to find without more sophisticated tools.

One of the oldest debugging tools is the core dump, usually taken after the program had made an error that upset the hardware. The entire contents of memory and all the registers would be printed in hex or octal, with very little to signify what was what. With the advent of high-level languages, the meaning of such dumps became more obscure, although they are still used [Blai 71, Gris 70, Kuls 71]. However the trend is to format dumps so that they relate to the source program. Stack dumps on the Burroughs B6700 are easy to follow when used in conjunction with the compiler option STACK (which is available for all major compilers). The B6700 also has a dump analyser for core-dumps which makes them much easier to understand. Selective and snap dumps are more useful to the high-level language programmer. Dumping suspect variables at carefully chosen places in the program will give a lot of information with relatively little output. [Ferg 63, Gain 69, Satt 72].

Tracing is another old debugging tool. There are many kinds of trace, but they all show where some trace condition or trap has occurred. Typical traps are storing to a variable (store trace), change of flow of control (flow trace) or reaching a given line of code (line or source trace). Traces are very useful as they can give a full history of program execution. They can, however, generate a lot of superfluous

output if they are not controlled [Grov 74]. Dynamically turning the trace on and off [Gris 70, Gris 73], tracing only when a dynamic condition is true [Blai 71, Ferg 63] or limiting the trace by a static condition or loop [Burr 74 A, Burr 74 F, Satt 72] will reduce output considerably. With interactive systems, the trace can even be turned on or off by the programmer [Bull 72, Kuls 71].

Program statistics are useful for deubugging and increasing efficiency. The simplest statistic is the execution time for the program. This in itself can often help to locate errors. The execution summary of SNOBOL 4 [Gris 73] is an extension of this. Burroughs Algol has a compiler option to print the time spent on each procedure of a program. MUSSEL and Algol W go one step further and give the execution count for each statement [Grov 74, Satt 72]. A different kind of statistic is the cross-reference listing, [Brow 73, Burr 74 A], which is particularly useful when looking through large programs.

All the tools mentioned so far are batch-oriented. With interactive systems a much greater range of tools can be made available. The basis of most interactive debugging systems is breakpointing. A breakpoint is a device for giving control to the programmer at an interactive terminal. The programmer can then converse with his program, see exactly what it is doing, and even correct it if it is wrong (section 1.4).

Interactive debugging in the days of machine-code programming consisted of stepping through the program instruction by instruction until something looked wrong. By using console switches a correction would be made and the process of detecting and locating bugs would continue. Stepping statement by statement is the high-level language equivalent, and it is just as useful [Gain 69, Pier 74, Burr 76 A]. This method can be extended by stepping several statements at a time [Pier 74] or running the program at an observable speed [Bate 69].

Bugs are usually detected some time after they occur, which makes them hard to locate. This is especially true of evanescent bugs which by Murphy's Law never seem to occur when they are being looked for. If, however, the program can be backed up to the point of error, the

problem is solved. Reversible execution can be implemented in two ways. The simplest is by checkpoints [Bate 74, Gris 71]. This requires little effort, but a forward execution from the checkpoint may differ from the original and evanescent bugs might not appear again. The other method is to record each change in the program as it happens [Bate 69, Davi 75, Zelk 73]. This takes a lot of process time, but using the history file for forward execution when possible will guarantee the same execution path will be used.

It is often useful to run a program again without changing its variables. This is an important feature of an incremental compiler called incremental execution [Bull 72, Rish 70]. Incremental compilers can also run the program from any point in the program, or even run an incomplete program. This greatly facilitates debugging as small sections of code can be tested independently of the rest of the program.

There is a right way and a wrong way to use any tool, and debugging tools are no exception. The programmer should get an idea of what is wrong with his program and carefully select his tools rather than apply battering-ram tactics. It will be cheaper, quicker and cause less headaches. Knowing what errors might occur is a start [Brow 73, VanT 74]; certain errors are best located with certain tools. For example, bad initialisation can be easily found by a store and/or fetch trace while looping errors can often be found by using execution counts.

1.4 Bug Correction

Once a bug has been located, it must be put right if the program is to work correctly. Bug correction usually consists of patching the source program and recompiling it, although only the offending subroutine may be recompiled and then bound or link-edited to the rest. In either case, the program must be run again from scratch. This approach is time consuming to the programmer and to the machine. The alternative is to correct the error at run-time.

When an exception condition occurs, the computer will detect it and terminate the program. However, if PL/I or Burroughs Algol 'ON' statements are used, the program can retain control. This will enable the program to make some correction or output suitable for debugging information. At any

rate, the program can continue execution, which may lead to finding other bugs.

Such interrupts are used for hardware and related errors, but software errors are not so easily detected or corrected. Recovery blocks are a solution whereby a section of code will ensure a condition, which is like a predicate in formal proofs [Rand 75]. If the main piece of code, the primary alternative, does not satisfy the condition, other alternatives are executed until the condition is satisfied or there are no more alternatives, which causes a fatal error. This allows the program to continue and give reasonable results even it if is not completely correct.

Conversing interactively with a running program, inspecting and changing its variables is a powerful tool [Barr 69]. If a programmer can watch his variables change value, he can actually see his program go wrong and possibly see why it went wrong [Balz 69]. In any case, correcting wrong values will temporarily patch the program. The ability to do this is the basis of most breakpoint debugging systems [Appendix A] and is an important advantage of time-sharing.

If the programmer has to correct the same error each time it is executed, debugging will take a long time. By making a run-time patch, he actually changes the program so that hopefully the error is fixed for that execution of the program. Unfortunately, run-time patches on most debugging systems often do not resemble the source language of the program and so are not permanent [Bate 74, Blai 71, Gris 70]. Also, they are usually very limited as to what they can do (assign only constants [Bate 74, Burr 74A], no conditional statements [Ashb 73]).

Incremental compilers solve all these problems. Patches have the same status as any other part of the program. They are in the same source language and suffer no restrictions as the same compiler is used for patches as for the rest of the program [Bull 72, Ryan 66]. When an error is located at run-time, a patch can be made, linked into the rest of the program and the same run continued as if nothing had happened. Editing the program at run-time is an important aspect of incremental compilers [Rish 70].

So far, the programmer suggests the correction, tries it and lets the computer find out if it works. Davis suggests that with inexperienced programmers, it is better for the computer to suggest the mistake as well [Davi 75]. When an error occurs, the computer backs up the program showing how the error was reached and what could have caused it. When the user decides on the cause, the computer explains how it could fix the mistake, checks with the user and does the fix.