SOFTWARE BUGS

Prior to the 1960s, most programs were made by small teams, usually consisting of a single person. Software was generally undocumented and errors could only be corrected by the original author. In those days, people concentrated mainly on the computer hardware, which was the primary limiting factor in computing. The main challenge in creating software was to squeeze the programs into small amounts of memory. Gradually, the cost of memory and other computer hardware dropped and at the same time size and complexity of software increased substantially. In 1961, the released software for the IBM 709 consisted of about 100 K words of program written by a small group of highly qualified people (1).

During the 1960s, it gradually became evident that the reliability of a computer system is largely determined by the reliability of its software components. The conventional belief became that there were always bugs in programs. In fact, the use of the term bugs to denote software faults is perhaps a form of psychological self-defense; everybody knows that the world is full of bugs and that little can be done about them. The process of eliminating bugs, known as debugging, was the next hurdle to overcome.

The following story describes the first program bug (2). Early in the history of computers (in 1945), when the Whirlwind I at the Massachusetts Institute of Technology (MIT) was first switched on, it failed to run. A frantic check of the wiring and hardware failed to indicate anything wrong. Finally, in desperation, it was decided to check the program, which was contained on a small strip of paper tape. The error was discovered in the programmers' Pandora's box, and a variety of bugs have been discovered by subsequent generations of programmers.

With the development of high-level languages and compilers, some people assumed that software bugs would disappear. However, this assumption ignored the fact that logic errors cannot be discovered by compilers because a compiler does not know what the programmer wants to do. Programs have continued to increase in size and complexity while keeping about the same level of bugs.

Writing a program is like writing a report. It requires a first draft (before debugging) and a final draft (after debugging). An important measure of a programmer's proficiency is the ability to find and correct the program bugs in an efficient manner. As programs, and interrelated sets of programs, became increasingly large and complex, more and more of the programmer's time was spent not in program design and coding, but rather in debugging and testing. While beginners may have a hard time locating and correcting their bugs, experienced programmers can do so more easily. Programmers are often trained in programming, but seldom are they trained in debugging. Debugging of a program usually takes more time and is more complicated than writing the program itself. It is therefore wise to spend more time in learning how to debug programs.

The presence of bugs in programs can be regarded as a fundamental phenomenon; the bug-free program is an abstract theoretical concept like the absolute zero of thermodynamics, which can be envisaged but never attained. Debugging is also dependent on the environment, including the machine, the language, the operating system, the problem, and the individual program. Thus, the study of bugs and debugging is an important undertaking.

SOME DEFINITIONS

Computer programming is used in the task of developing a software. This programming is not difficult, but it must be done with care and involves much more than just writing instructions. To create software that allows us to use the computer effectively as a problem-solving tool, several steps must be carried out. These steps include defining the problem, planning a solution algorithm, coding the algorithm, checking the program (debugging and testing the algorithm), and completing the documentation. After a problem solution has been planned and coded accordingly, the programmer must make certain that the program performs as intended. This task is part of the programmer's responsibility for complete and comprehensive program checking. A major concern in this respect is the issue of isolating, identifying, and correcting bugs. This step requires special care in order to avoid creating new bugs when correcting the existing ones. In general, as the size and the complexity of a program increase, a higher portion of the programmer's time is spent in debugging and testing, as compared to the actual design and coding.

Software professionals emphasize that program checking should begin in early stages of the software development. Certain types of errors can be detected and removed at the time of problem definition, while some others can be detected in the process of formulating the solution algorithm and coding the corresponding program. Concise and accurate documentation is a vital task throughout the software development cycle and must be carried out on a continuing basis.

A program failure is caused by an error, which itself is the consequence of a fault (a slang expression for a software fault is *bug*). Figure 1 illustrates the concepts of fault, error, and failure, which are formally defined as follows:

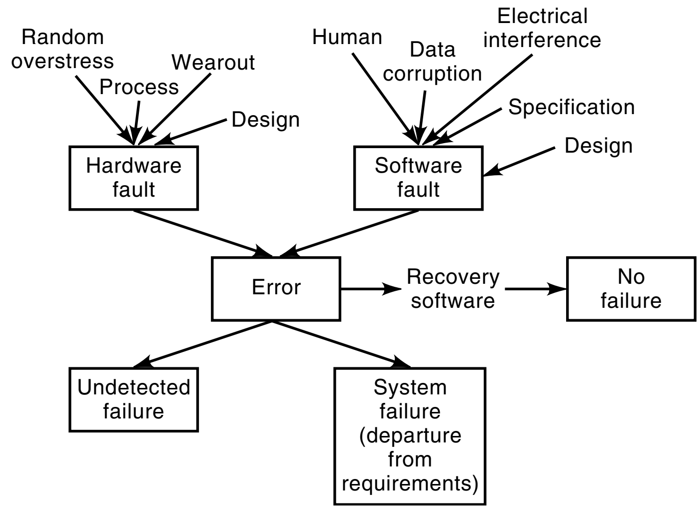

Figure 1. Concept of fault/error/failure. A fault (bug) may lead to an error. An error may propagate to become a failure if the system does not contain some error recovery logic capable of dealing with and minimizing the effect of the error. A failure, whether hardware- or software-related, is the termination of the ability of an item to perform its specified function.

1. *Faults* may occur in both hardware and software. Software faults will arise when a problematic part of the code is executed (with a particular set of values for the relevant parameters resulting in the erroneous condition) or because of corruption due to some outside influences, such as memory corruption. Software faults are conditions that may lead to an error in the system. These faults may be due to ambiguities, omission in the logic structure of the program, or hardware conditions, which can cause software corruption. A fault (bug) may lead to an error and eventually to a system failure. However, the presence of a software fault does not necessarily guarantee that an error or a failure will ensue. A long time may elapse before that specific portion of the code is used under the circumstances that lead to a failure.
2. *Errors* occur when the software in the system reaches an incorrect state. An error is caused by a fault in the program or by an outside interference. An error may propagate to become a failure if the system does not contain some kind of error recovery logic capable of dealing with the specific error. *Error recovery software* may prevent the propagation of an error.
3. *Failure* is the termination of the ability of an item to perform its specified task. Software failures are, in fact, errors that, due to the complexity of programs, do not always become evident immediately. Unlike hardware failures, there may not be any physical change associated with a software failure that causes a functioning unit to cease its normal operation.

The actual source of faults may be the requirement specification, the design, or the implementation. There is evidence that the majority of errors (over 60%) are committed during the requirement and design phases. The remaining 40% occur during coding. The more complex the system, the more faults are likely to initiate from ambiguities and omissions in the specification stage.

CATEGORIES OF BUGS

Assuming that the input data is correct, we can broadly divide computer bugs into three different categories. These are bugs related to hardware, systems software, and the pro-

Table 1. Percentage of Bugs' Category

Category of Bugs	Percentage
Hardware	1%
Systems software	9%
Programming	90%

gramming itself. A rough estimate of the relative incidence of these different types of bugs is given in Table 1.

Hardware Bugs

Hardware bugs are mercifully rare nowadays and are often easily detected. However, an intermittent hardware bug can be extremely difficult to detect and may persist for a long time before it can be pinned down. Usually, the software is blamed first, and hardware is checked as the last resort. Therefore, these types of bugs are inherently costly and time-wasting.

Systems Software Bugs

In the following, we discuss the implication of system software bugs, which vary widely with the type of software. The types of system software bugs we distinguish are as follows:

- *Operating Systems.* Operating systems are immensely powerful and complex, so the chances of them being bug free are minimal. Operating systems are the most-used pieces of software, so producers take considerable care to check them carefully; however, the presence of bugs in them is still certain.
- *Compilers.* After operating systems, compilers are probably the second most-used software in an installation, so most manufacturers take a good deal of care to make them as bug free as possible. The user is usually completely unaware of what actual machine-code instructions are generated by a compiler. Therefore, if a failure is encountered at some point during execution, the programmer has to assume that the corresponding bug is his or her own fault. The task of debugging becomes much more complicated if the compiler has indeed generated an invalid object code from an originally valid source program. All compilers have some restrictions, which may not even be described in the manual. If checking of these restrictions in conjunction with a particular bug is feasible, it should be tried so as to sidestep the bug.
- *Utility and Application Packages.* Like compilers, various systems and applications software packages supported by an installation may not be entirely bug free. However, at the same time, when a bug is encountered while using these packages, the user should first assume that the bug lies in his or her part of the code. This class of software makes a large volume of the software available on any installation and, for the lack of any better term, is lumped into a single category called systems and applications software. Most installations also have a collection of internal routines, macros, library procedures, and so forth, the use of which is highly recommended. Great care is usually taken before releasing such subroutines for general use. However, like any other software, these should not be considered bug free.
- *Programs Written By an Outside Agency.* Strictly speaking, programs written by a software house for an installation (to its own specifications) should not be "black boxes" to the installation's maintenance programmers. Maintenance programmers should be provided with adequate technical documentation to make future debugging possible.

Programming Bugs

By far the most frequent and complicated bugs are due to mistakes in the program itself. These bugs range from specification to implementation. Table 2 summarizes these kinds of bugs.

Errors in Problem Definition. It may happen that once the program is written, the user finds out that the results are not as expected. This can be because the programmer and the user have not understood each other properly, or because the user did not exactly know what he or she wanted. In this case, the incorrect program may help the user and the programmer better understand the underlying problem, in which case their efforts will not be completely wasted.

Sometimes only when incorrect results are generated can the original problem be carefully redefined. An improper problem definition may result in a program that provides a correct solution for an incorrect problem. In such a case, a new definition of the problem may need to be formulated, which requires a great deal of fresh implementation effort.

Incorrect Algorithm. Once the problem is correctly defined, the programmer searches for an algorithm or method to solve the problem. Unfortunately, the programmer may choose a poor or even an incorrect algorithm, in which case he or she has to repeat the whole process at some later point.

Errors in Coding. There is a large variety of errors that fit this category:

Syntax errors are due to improper use of the language statements. These are often detected and flagged by the compiler.

Logic errors are another type of error in coding. Most programmers introduce certain types of errors, which they tend to repeat over and over. In such a case, it is advisable to keep a list of such commonly encountered errors. In other words, a programmer with long experience in debugging can think of and prepare a personal list of his or her typical errors. This list can then be used as a checklist during the debugging indicating what to look for once a new bug is encountered. Common examples of these types of bugs include using illegal subscripts,

Table 2. Common Programming Bugs

1. Errors in problem definition	Correctly solving the wrong problem
2. Incorrect algorithm	Selecting an algorithm that solves the problem incorrectly or poorly
3. Errors in coding	Incorrect program for the algorithm

writing conditional jumps to a wrong place, or counting from one when counting should start from zero. These types of errors are particularly common if one habitually programs in two or more languages. Note that logic errors are not syntax errors and will still be present after syntax checking is complete. The following is a partial classification of logic errors according to their types:

- *Loops* (e.g., wrong number of loop cycles)
- *Data and input/output* (e.g., failure to consider all possible data values)
- *Variables and arithmetic operations* (e.g., using an incorrect variable name, or a spelling error causing the use of a wrong variable)
- *Arrays* (e.g., transposing the subscript order, or index out of range/bounds)
- *Subroutines* (e.g., use of incorrect parameter values in a subroutine call)
- *Character strings* (e.g., declaring a character string with the wrong size)
- *Logical operations* (e.g., failure to provide a properly matched ELSE clause in a nested IF . . . ELSE statement)

The aforementioned bugs are mostly detected in the early phase of debugging. Beyond these, there exists a whole class of more complicated bugs that belong to later stages of debugging. We refer to this class of bugs as *special* bugs. These are sophisticated errors that are difficult to locate. Here are some examples of such bugs:

- *Semantic Bugs.* These bugs are caused by the failure to understand exactly how a command works. An example is to assume that arithmetic operations are automatically rounded. Another example is to assume that a loop will be skipped if the ending value of the loop variable is smaller than the initial value.
- *Semaphore Bugs.* This type of bug is exemplified by the situation when a process A is waiting for an event that can only be caused by a process B while the process B is waiting for an event that can only be caused by the process A. This type of bug usually emerges when running large concurrent systems such as an operating system.
- *Timing Bugs.* These bugs can develop when two operations depend on each other in a temporal sense. For example, suppose the operation A must be completed before another operation B can start. If operation B starts too soon, a timing bug may appear. Timing bugs and semaphore bugs are also known as *situational bugs.*
- *Evanescent Bugs.* Another type of nasty bug that is intermittent is called an evanescent bug. This is a bug that may appear and then disappear for a period of time. This includes bugs that will not reappear even when the program is rerun with identical data on the same machine. An example of this type of bug is a program switch that has not been initialized but usually is correct due to the tendency of the machine to have a zero in that particular location.

PREVENTING BUGS

Debugging is often the most costly part of software development. Thus effort should be made to prevent bugs. There are a few rules that, if followed by software developers, will help to eliminate some common bugs:

- *Avoid Questionable Coding.* It is better to avoid using advanced features unless one has made certain that they do perform as expected. One should not try to fool the compiler or the operating system. Compilers and operating systems are very complicated, and it may be possible to find a situation in which one can violate a language defined rule and still get correct results. However, such actions should be avoided. This type of bug can be very difficult to find, specifically if the program has been used for a while.
- *Avoid Dependence on Defaults.* All programming languages have some defaults, which the compiler assumes. The use of these defaults saves work for the programmer but can be dangerous because computer manufacturers occasionally change the defaults. Different machines have different defaults, and if it is desirable to maintain portability of programs, it is best to avoid using too many defaults.
- *Never Allow Input Data Dependency.* One should not allow a program to depend on whether the input data is in a specific form or is within a restricted range. Instead, input data should be checked within the program to make sure that they are correct. If data are not checked at input, the program may periodically be found to have mysterious failures. Such bugs usually result in a reputation of unreliability for the program and the programmer.
- *Check for Completeness of Logic Decisions.* For example, if data are supposed to take a value of one or two, one should not just check for the value of one and then, if false, automatically assume a value of two. This will overlook the pathological cases that may be present. Instead, the data should be examined for the value of one; then, if not true, the data should be examined for the value of two. If it is neither one nor two, then one should provide code for the pathological case (that is, usually an error message or halt).
- *Employ a Debugging Compiler.* The compiler in use greatly affects the amount of debugging needed. A debugging compiler checks for more complicated errors as compared to a regular compiler. A good debugging compiler can often reduce the debugging time. Syntax is more carefully examined and the interaction of commands is checked. More important, numerous checks are done during execution of the source program. Uninitialized variables, out-of range subscripts, and illegal transfers are flagged during execution. Obviously, all this additional checking requires extra time, so execution time is usually much slower.

TESTING VERSUS DEBUGGING

Many programmers confuse the debugging and testing stages of the program development and treat these two activities as equivalent. However, these are two distinct and different activities (3). Testing is the dynamic execution of the software under controlled conditions with a sample input. Testing is done for two purposes: (1) to identify errors (during development), and (2) to give confidence that the system is working (during acceptance testing). If the testing stage provides an evidence of any program failure, then the debugging stage will follow. The process of locating and correcting errors in software is known as debugging, so called because one of the earliest faults found in a computer was a suicidal moth (bug) trapped in a relay, which caused incorrect operation of the software. Debugging always starts when some evidence of program failure is observed.

Often, after tests have been run, the program will fall back to the debugging stage. Testing determines that an error exists; debugging first localizes and then removes the cause of the error. Thus, there is some overlap between these two stages. Programming time should be allotted for both stages in order to emphasize that both of them are necessary.

THE DEBUGGING PROCESS

Debugging is the procedure of iteratively isolating the location and the cause of a failure (not withstanding the fact that one might get lucky and find it on the first pass through the debugging procedure). Debugging is performed after executing a successful test case indicating a failure. In more concrete terms, debugging is a two-part process; it begins with some indication of the existence of an error (e.g, the results of a failed test case), and it is the activity of (4)

1. Determining the exact nature and location of suspected error within the program
2. Fixing or repairing the error

Usually, determining the cause of a failure requires much more effort as compared to setting up the corresponding test case (revealing the failure). Debugging, then, should be of major importance to anyone concerned with improving programming productivity. The correction usually consists of making a change to software and its associated documentation, but it can also consist of changes to the test documentation, user documentation, or operational procedures.

Novice programmers often believe that a program needs to be debugged only once. That is, when the program works nicely in conjunction with a selected set of data, they assume that it will work for all other data as well. They will be often surprised when, after using and believing the results for several runs, they find out that the program is producing an obviously incorrect output. This means that, in reality, a program may continue to require debugging throughout its life.

There are two general approaches to debugging. In the first approach, debugging is achieved once the program is complete. In this case, either a great deal of programmer time is spent trying to avoid and detect bugs manually, or the machine's help is sought in detecting bugs. The choice between the two alternatives is governed by the amount of machine time available. There is a natural tendency to push most of the debugging work off on the machine. If machine time is available, this is wise since the machine (equipped with an appropriate debugger) may be more effective. There is, however, a second approach to debugging. In this approach, debugging overlaps with the writing stage of programming. Some programmers prefer to write a few lines of code and then test them immediately to make sure that they work properly. Programmers who program this way are writing, debugging, and testing all at the same time.

STAGES OF DEBUGGING

As already mentioned, the debugging process begins with the execution of a test case for which the results are assessed and a lack of correspondence between expected and actual values is encountered. The debugging will always have one of the following two outcomes: (1) The cause of the error will be found, corrected, and removed; or (2) the cause of error is not found, in which case the person performing debugging may suspect a cause, design a test case to help validate his or her suspicion, and work toward error correction in an iterative manner. This means that during debugging we encounter errors that range from mildly annoying cases (e.g., an incorrect output format) to catastrophic (e.g., a system failure). The following typical situations are possible during the stages of debugging:

- *Case 1: Program Outcome does not Match the Desirable Specification.* A failure is actually a behavior that does not match the program specification. Thus, one should first consult the specifications themselves to determine whether they are clear enough and to consider the possibility that the error is in the specification rather than in the implementation. This means that when our objective is to prevent errors, we must direct our attention to the start of the program development process rather than to the end of it. In other words, a reasonable first step to debugging is to verify the completeness and accuracy of the problem definition.
- *Case 2: Program Terminates Prematurely.* The program compiles properly, starts execution, provides some output, and then terminates earlier than expected. In this case, since some output is being produced, regular debugging techniques can be applied.
- *Case 3: Incorrect Answers.* The program runs but produces incorrect answers. Experienced programmers always consider themselves lucky when this stage is reached. This probably indicates that the program is basically sound and the logic is almost correct.
- *Case 4: An Infinite Loop.* This error is usually not very difficult to find. If you cannot spot the loop immediately, simply add print statements before and after suspected loops. Do not put print statements in the loops; otherwise, thousands of lines of output will usually appear. The print statements will provide output that will indicate which loop is entered but never exited. Another common situation where a program may appear to be in an infinite loop may actually arise due to indefinite wait caused by the lack of expected input or some other event (e.g., a message from some other process).

DEBUGGING ALGORITHM

It is evident that a computer can neither construct nor debug programs without being told, in one way or other, what problem is supposed to be solved and some instructions on how to solve it. No matter what language we use to convey this information, we are bound to make mistakes. This is not because we are sloppy and undisciplined, as advocates of some program development methodologies may say, but because of a much more fundamental reason: We cannot know, at any given point in time, all the consequences of our current assumptions. A program is indeed a collection of assumptions, which can be arbitrarily complex, and the resulting behavior is a consequence of these assumptions. As a result, we cannot, in general, anticipate all the possible behaviors of a given program. It follows from this argument that the problem of program debugging is present in any programming or specification language used to communicate with the computer and hence should be solved at an abstract level. In particular, we attempt to formalize and develop algorithmic answers to the following two questions:

1. How do we identify a bug in a program that behaves incorrectly?
2. How do we fix a bug, once it is identified?

An algorithm that solves the first problem is called a *diagnosis algorithm,* and an algorithm that solves the second is called a *bug-correction algorithm.* To debug an incorrect program, one needs to know the expected behavior of the target system. Therefore, we assume the existence of an agent, typically the programmer, who knows the target program and may answer queries concerning its behavior. The programmer, in turn, may have gained this information from the specifications.

A diagnosis algorithm and bug-correction algorithm can be integrated into a *debugging algorithm,* following the scheme in Fig. 2. A debugging algorithm accepts as input a program to be debugged and a list of input/output samples that partly define the behavior of the target program. It executes the program on the input samples; whenever the program is found to return an incorrect output, it identifies a bug in it using a diagnosis algorithm, and fixes it using the correction algorithm.

DEBUGGING PRINCIPLES

A set of debugging principles, many of which are psychological in nature, is discussed in the following section. Many of these principles are intuitively obvious, yet they are often forgotten or overlooked. Since debugging is a two-part process (locating the error and then repairing it), the set actually consists of two subsets (5):

Error-Locating Principles

- *Think.* We know that debugging is a problem-solving process. The most effective method of debugging is a mental analysis of the information associated with the error symptoms. An efficient debugger should be able to pinpoint most errors prior to the execution of the program.
- *If You Reach an Impasse, Sleep on It.* The human subconsciousness is a potent problem solver. What we often refer to as inspiration is simply the subconscious mind working on the problem while we might be consciously doing something else, such as eating, walking, or watching a movie. If you cannot locate an error in a reasonable amount of time, drop it and work on something else. After "forgetting" about the problem for a while, either your subconscious mind will have solved the problem or your conscious mind will be clear for a fresh reexamination of the symptoms.
- *If You Reach an Impasse, Describe the Problem to Someone Else.* By doing so, you will probably discover something new. In fact, it is often the case that by simply describing the problem to a good listener, you will suddenly see the solution without any real assistance from the other party.
- *Avoid Experimentation, Use It Only as a Last Resort.* The most common mistake made by novice debuggers is attempting to solve a problem by making experimental changes to the program (e.g., "I don't know what is wrong, so I will change this statement and see what will happen."). This totally haphazard approach cannot even be considered debugging; it represents an act of blind hope. Not only does it have a miniscule chance of success, but it often compounds the problem by adding new errors to the program.

Error-Repairing Principles

- *Errors Tend to be Clustered.* Where one bug exists, there is likely to be another, so when one finds an error in a section of a program, the probability of the existence of another error in that specific section is higher. When repairing an error, examine its immediate vicinity for anything else that looks suspicious. As the complexity increases, the defect (bug) density increases. In general 80% of all bugs in a program are located in the 20% most complex modules.
- *Fix the Error, Not Just a Symptom of It.* Another common improper act is to repair the symptoms of the error, or just one instance of the error, and not the error itself. If the proposed correction strategy does not match all the clues about the error, one may end up fixing only a part of the error and not all of it.
- *The Probability of the Fix Being Correct is Not 100%.* A new piece of code that is added to a program to fix an error can never be assumed to be perfectly correct. In general, corrections are much more error prone than the original code itself. One implication is that corrections must be tested, perhaps more rigorously than the original program.

```
read P, the program to be debugged.
repeat
    read the next input/output sample.
    while P is found to behave incorrectly on some input do
        identify a bug in P using a diagnosis algorithm;
        fix the bug using a correction algorithm.
    output P.
until no samples left to read.
```

Figure 2. A scheme for a debugging algorithm. It accepts as input a program to be debugged and a list of input/output samples. Whenever the program is found to return an incorrect output, the scheme requires identification of the bug using a diagnosis algorithm, and a fix for the bug.

- *The Probability of the Fix Being Correct Drops as the Size of the Program Increases.* In other words, the ratio of errors due to incorrect fixes versus original errors increases in larger programs. Experience has shown that in a large program, on the average, one of every six new errors discovered is due to prior corrections to the program.
- *Beware of the Possibility That an Error Correction May Create a New Error.* Not only does one have to worry about incorrect corrections, but one has to worry about seemingly valid corrections that may have an undesirable side effect leading to a new error. One implication is that not only does the error situation have to be tested after the correction is made, but one must also perform regression testing to make sure that a new error has not been introduced.
- *The Process of Error Repair Should Put the Programmer Back Temporarily in the Design Phase.* One should realize that error correction is a form of program design. In other words, whatever procedures, methodologies, and formalism were used in the design process should also apply to the error-correction process.

DEBUGGING APPROACHES

Regardless of the approach taken, debugging has one overriding objective: to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition, and luck. In general, the following categories for debugging approaches are commonly used (6):

1. *Debugging by Brute Force.* The most common method of program debugging is the rather inefficient brute force method. Perhaps the reason for its popularity is that it requires little thought. However, the brute force method is usually the most inefficient and unsuccessful approach to debugging. This method can be partitioned into at least three categories:
   - Debugging with a storage dump, whereby the programmer prints out the whole or a part of the memory image of the program at a certain point during the execution. The programmer then attempts to locate the error by analyzing the values of data or stack variables.
   - Debugging via insertion of print statements in those parts of the program where the bug is expected. These statements are generally used to print the values of those variables that may be helpful in locating the error.
   - Debugging via complete reliance on automated debugging tools that may allow a programmer to execute the program under the controlled conditions, stop the program at certain points, examine values of data variables, and so on.

   The general shortcoming of these brute force methods is that they ignore the process of thinking. It is our contention that most errors can be located by careful thinking, in many cases without even further using the computer. Some instances of such thought process are explained in the following list item.
2. *Debugging by Induction.* In an induction process, one proceeds from a particular point to the whole. That is, by starting with the clues (symptoms of the error, possibly gathered from the results of one or more test cases) and looking for relationships among them, one can often locate the error. The induction process is illustrated in Fig. 3. The steps are as follows:
   - Locate the pertinent data.
   - Organize the data.
   - Devise a hypothesis.
   - Prove the hypothesis.
3. *Debugging by Deduction.* The process of deduction, illustrated in Fig. 4, is a process of proceeding from some general theories or premises, using the process of elimination and refinement, to arrive at a conclusion (the location of the error). The steps are as follows:
   - Enumerate the possible causes or the hypotheses.
   - Use the data to eliminate possible causes.
   - Refine the remaining hypothesis.
   - Prove the remaining hypothesis.
4. *Debugging by Backtracking.* An effective error-locating method for small programs is to backtrack the incorrect results through the logic of the program until one discovers the point where the logic went astray.
5. *Debugging by Testing.* The last "thinking-type" debugging method is the use of test cases. In general, one can consider two types of test cases: test cases for testing,

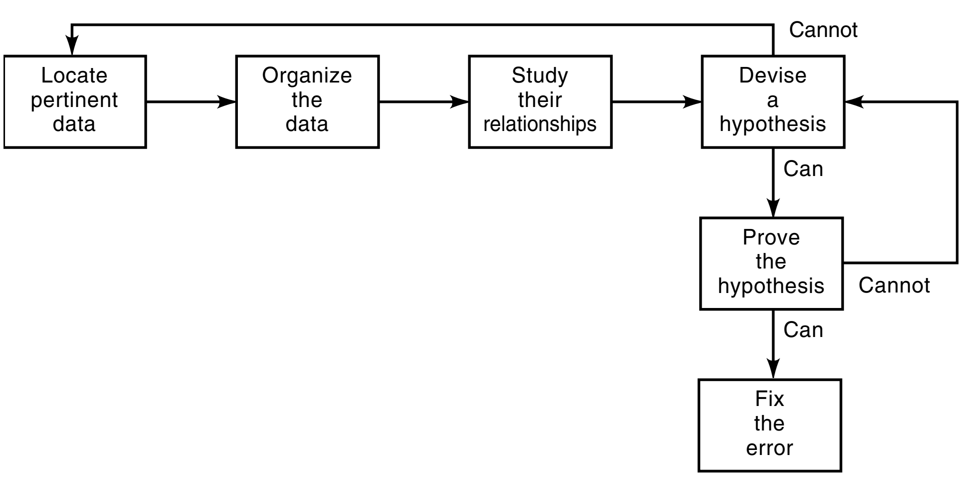

Figure 3. Most errors can be located by careful thought. One such thought process is induction. The first step is the enumeration of what the program did correctly, and what it did incorrectly. The second step is the structuring of the pertinent data to allow one to observe patterns. The next two steps are to study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error. A hypothesis is proved by comparing it with the original clues or data, making sure that the hypothesis completely explains the existence of the clues, which is the last step.

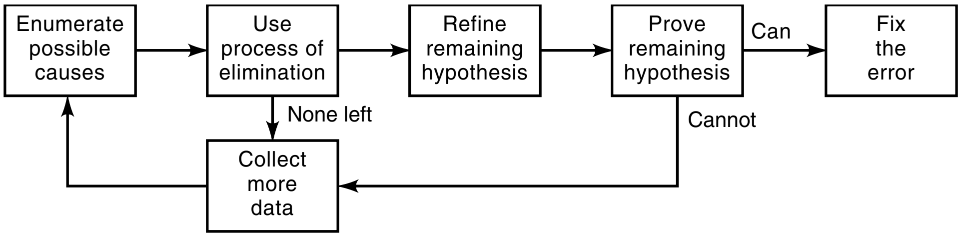

Figure 4. The process of deduction. The first step is to develop a list of all conceivable causes of the error. By a careful analysis of data, one attempts to eliminate all but one of the possible causes. The available clues are used to refine the theory to something more specific. The last vital step is identical to the last step in the induction method.

in which the purpose is to expose a previously undetected error, and test cases for debugging, in which the purpose is to provide information useful in locating a suspected error.

6. *Debugging by a Combined Approach.* As a final remark, we note that the preceding approaches are not mutually exclusive, and most often programmers employ a proper combination of them.

USE OF DEBUGGING AIDS

Debugging aids are the tools that a programmer uses to debug a program. As with tools of any kind, they must be used in the proper place and in the correct way to give acceptable results. A good debugging tool should be flexible and easy to use.

A repertoire of debugging aids is a useful source of help during debugging. But such tools seldom relieve the programmer from constructing his or her own debugging aids. The often effective debugging aids seem to be those that are written into the program while writing the original program (7). Common examples of debugging aids employed by programmers include the following:

- *Dump* is a record of information at a given time of the status of the program. This is usually provided in machine language and is of limited use for several reasons. The main reason is because it is difficult to relate the dump to your program. It requires the programmer to understand machine language and be able to relate machine language to the high-level programming language in use. In addition, if the compiler optimizes high-level code, it becomes even more difficult to use the dump even if machine language is known. A highly optimizing compiler can entirely rearrange the operations in a program, thus making a dump almost useless. Since the information provided in a dump is not in a form that can be used, there has been a trend to provide debugging aids, which provide debugging information in a form more suitable for use.
- *Trace* is a record of the path of execution of the program. It can be used to see if the program is being executed in the same sequence as the programmer intended and if the variables have the desired values stored in them. There are usually three types of traces:
  - *Flow.* The first type traces the flow of control of the program. That is, it usually prints statement labels as they are passed during execution.
  - *Variable.* This type of trace prints variable names and values. Every time a variable changes its value, the variable label and its new value are printed. These traces are designed so that, instead of printing out all variables, only a selected subset of them is monitored and printed.
  - *Subroutine.* The third type of tracing involves tracking subroutine calls. This becomes very useful in a program that calls many subroutines. Every time a subroutine is called, the name of the subroutine is printed; and when a return from the subroutine is executed, a return message is printed.

  Traces will often provide all the information needed to locate a bug in a program. But their weakness is that they can easily provide too much information (that is, thousands of lines of output). The second disadvantage is that, because of the great amount of information monitored and provided, traces are usually quite costly in machine time. A full trace can easily increase execution time by a factor of 10 to 40. Thus, in order to overcome these difficulties, flow traces are usually designed so they can be turned on and off. That is, they can be turned on just for the section of the program that needs to be traced and turned off for the other sections.
- *Subscript check* monitors the validity of all subscripts used with the named array by comparing the subscript combination with the declared bounds of the array. If the subscript falls outside the declared range, an error message is printed. It is usually possible to monitor all, or just a subset, of the arrays.
- *Display* allows the user to select the exact place in the program when the variable value is to be printed. This allows a much more selective printing than the variable trace. In addition, the display command usually prints the variable name along with the variable value. This provides labeled output automatically.

BASICS OF DEBUGGERS

A debugger is a tool to help track down, isolate, and remove bugs from software programs (8). Debuggers are tools to illuminate the dynamic nature of a program. They are used to understand a program, as well as to find and fix its defects. Debuggers are like a magnifying glass, the microscope, the logic analyzer, the profiler, and the browser with which a program can be examined. Debuggers are quite complex pieces of software that also require an exceptionally close cooperation with and intimate knowledge of the operating system. Here are some basic facts about debuggers:

- *What Are They?* Debuggers are software tools that help determine why a program does not behave correctly. They help a programmer in understanding a program and then in finding the cause of its defect. The programmer can then repair the defect and so allow the program to work according to its original intent. A debugger is a tool that controls the application being debugged so as to allow the programmer to follow the flow of program execution and, at any desired point, stop the program and inspect the state of the program to verify its correctness.

- *Who Uses Them?* Typically, the original developer uses a debugger, but later a maintainer, a tester, or an adapter may also use it. A debugger can also serve as a useful way for someone unfamiliar with a piece of software to get up to speed on that code in a preparation for maintenance or expansion of the code.
- *How Are They Used?* Debuggers are used by rerunning the application, sometimes after a special compilation that prepares them for debugging, in conjunction with the debugger tool itself. The debugger carefully controls the application using special facilities provided by the underlying operating system to give the user fine control over the program under test. The user controls execution using commonly found debugger features such as *breakpoints* and *single-step* executions. The state of the program is examined until the cause of the defect is detected; then the programmer can attempt a fix and begin to search for any other defects.
- *Why Are They Used?* Debuggers are a necessary part of the engineering process, particularly when dealing with even moderately complex software systems. All interactions cannot be predicted, specifications usually are not written to the level of programming details, and implementations is an inherently difficult and error-prone process. As software gets more complex, debuggers become more and more important in tracking down problems.
- *When Are They Used?* First, debuggers are used at program inception time, when only part of the implementation of a design is complete. Second, when an identifiable module or subsystem is completed and ready for use, a debugger can help to make sure this component is ready for integration with the other components. Third, as testing process progresses on a complete program and uncovers new defects, the debugger becomes increasingly important because the program's bugs tend to get more difficult to detect and isolate over time. Fourth, debuggers are used as changes and adaptations are made to existing programs that introduce new complexities and therefore destabilize a previously working code.

SOFTWARE ENGINEERING PERSPECTIVE ON DEBUGGING

Structured programming can be used to model a large system as an evolving tree structure of nested program modules, with no control branching between modules except for module calls defined in the tree structure. By limiting the size and complexity of modules, unit testing and debugging can be done by systematic reading and by executing modules directly in a evolving system in a bottom-up testing process. We are interested in writing programs that are highly readable, whose major structural characteristics are given in a hierarchical form and are tied in closely to functional specifications and documentation. In fact, we are interested in writing programs that can be read sequentially in small segments such that each segment can be literally read from top to bottom with complete assurance that all control paths are visible in the segment under consideration.

Program design and the concept of "building" a program are terms that have now almost completely taken over the plain "writing" a program. The use of the terms *design* and *build* illustrates that engineering ideas and disciplines have now entered the programming world. Broadly speaking, this approach says that a software system or program should be treated like a piece of machinery. Therefore, for it to run smoothly, parts of it should be easily exchangeable, it should be easy to test, and so on. Thus, these features put a lot of emphasis on modularity, robustness, and testability.

All programmers nowadays adopt a modular approach to a large degree. No one admits to writing large, monolithic programs. When a program is broken down into small modules and each is specified separately, then clearly more thought will go into the detailed design work. In addition, smaller units mean less complexity and so should be easier to test. Also, having modular programs helps control coupling and the management of the interfaces. In the following, we first describe the spectrum of possibilities and give our definition of modular programming:

- *Monolithic.* The program is written in one large block of coding and may only be compiled and tested as one entity; only one programmer can write it.
- *Monolithic But of Modular Construction.* The program is written as a number of defined subroutines (perhaps written by several people) with a short "control program," which binds together the sections. The program may only be compiled as a whole but, by careful use of test aids, could be tested routine by routine.
- *Modular.* The program is written as a number of independent modules that are coded, compiled, and tested individually and then are brought together to form the whole program.

The best approach to program development involves looking first at the overall function to be accomplished by a program and then dividing that function into some lower levels, or subfunctions, each of which can be designed, coded, and tested with ease. The goal of this approach is its simplicity. It is based on certain interrelated improved programming technologies: top-down development, modularization, and structured programming. Programmers who follow the top-down approach to program development should not find themselves confronted with long, complex sections of unverified code. Although there are no absolute size limitations, individual modules are kept small in size, and unnecessary complexity is avoided by separating identifiable functions in independent parts. These parts are checked out as they are completed, over time, until a fully integrated program or system of programs is produced.

In summary, if a program is split into modules, which are written and tested separately and are only brought together when they have all been tested individually, then that is modular programming.

DEBUGGING VERSUS PROVING PROGRAM CORRECTNESS

It has been suggested that one way to eliminate the need for testing and debugging is to provide a correctness proof of the program. Given the current state of the art, techniques for proving the correctness of a program depend heavily on assertions, axioms, and theorems. This relates to the idea that, since a program is simply an algorithm by which symbols are manipulated, it should be possible to verify the correctness of the algorithm by a mathematical proof. As Naur and Randell say (9): "[When] you have given the proof of correctness, . . . [you] can dispense with testing altogether."

Investigation has shown that the difficulty of proving the correctness of a program is closely related to its complexity and to the number of interactions between its component parts. One of Dijkstra's hopes in developing structured-programming concepts was that automated proofs might be easier to develop for programs expressed in structured form. Although some progress has been achieved toward automating the proof process, it is still not possible to apply those techniques to software systems of a realistic size and complexity. In conjunction with the preceding quotation of Naur and Randell, Goodenough and Gerhart (9) recall a simple text formatter program described and informally proven correct by Naur, and they find seven bugs in it. Three of those bugs could be detected immediately by running the program on a single example. So they comment, "The practice of attempting formal or informal proofs of program correctness is useful for improving reliability, but suffers from the same types of errors as programming and testing, namely, failure to find and validate all special cases relevant to its specification, design, the program and its proof. Neither testing nor program proving can in practice provide complete assurance of program correctness."

Gerhart and Yelowitz (9) discuss the fallibility of some of the methodologies that claim to eliminate or reduce the need for debugging. They consider three types of errors—errors in specifications, errors in systematic program construction, and errors in program proving—and provide instances of each of these errors selected from published articles. Concerning errors in specification, they conclude, "These examples clearly show that specifications must be tested in much the same way that a program is tested, by selecting data with the goal of revealing any errors that might exist."

A program can be proven correct formally only with respect to another formal description of its intended behavior. This observation suggests that even if the effort in program verification succeeds, it does not solve the problem of program debugging, but simply reduces it to the problem of debugging specifications. If the problem of debugging specifications has not yet revealed itself as a serious one, it may be because there has been no intensive use of formal specifications in full-scale programming tasks. From an abstract point of view, however, a specification language that has a partial decision procedure is just another programming language, and for any programming language there is a complex programming task for which there is no simple, self-evidently correct program. As soon as complex specifications are used, there will be a need to debug them.

STATE-BASED APPROACH TO DEBUGGING

There is an alternate and significant view of program debugging called, the *state-based* approach. In this approach, the dynamics of the program under development (the target program) are observed from the viewpoint of *program states* (i.e., the values of the program-defined entities, and the point reached by the program *control flow*). Thus, the state of a sequential program can be characterized simply by the value of the program counter and the memory image of the program data. The *state history* is the record of the program states expressed in terms of the values assumed by the program-defined entities. The *flow history* is the record of the program state expressed in terms of the path followed by the program control flow. From this viewpoint, debugging techniques can be classified into two categories (10):

- *Tracing techniques* are based on the gathering and recording of portions of given behavioral aspects of the target program at specific execution steps. State and flow traces can be collected, which contain information on the program state history and the program flow history, respectively.
- In *controlled-execution techniques,* the user monitors the behavior of the program interactively, by means of break traps (also called breakpoints). When the process generated by the execution of the program enters the break state, the user examines and possibly alters the state of the program as well as the layout of the debugging experiment, dynamically.

These debugging techniques can be applied to any specific debugging approaches, such as deductive or inductive or a combination of approaches, described earlier. For example, once the existence of a bug has been revealed, the programmer forms one or more hypotheses about its cause. The program is executed with additional test data in order to collect more information concerning the error. The various hypotheses can be derived either by induction (which entails the differences between the unsuccessful and successful test cases) or by deduction (by using a list of possible theoretical causes for the suspected error). In either case, the program should be tested on the simplest input pattern that might prove or disprove each hypothesis. When the bug is located, appropriate corrections are determined and verified by repeating the tests. The process is iterated until a valid solution is found. To locate the program error, it may be necessary to exclude systematically parts of the program that have been demonstrated not to contain the bug, thus narrowing the code portion to be tested. This can be done by examining intermediate results using tracing or controlled-execution techniques.

DEBUGGING OF CONCURRENT PROGRAMS

A *concurrent program* consists of a set of sequential processes whose execution can overlap in time (i.e., a process can begin its execution before a previously started process has terminated). The processes may be multiprogrammed on the same processor, or they may be executed in parallel on different processors. They can be either independent or interacting, and interactions may take place for

- *competition,* to obtain exclusive access to shared resources
- *cooperation,* to exchange information and achieve a common goal

Competition imposes *mutual exclusion* on access to shared resources. For instance, one process must not be allowed to alter the value of a shared variable while another process is examining this variable. Cooperation places precedence constraints on the sequences of operations performed by the concurrent processes. For example, if a process has to use some data produced by another process, the former must wait for the latter to produce those data. Interprocess communications may occur via shared variables or message passing. In a shared variable environment, processes access some common memory. In a pure message-passing environment, however, processes do not share memory. Instead, interprocess communication and process synchronization are achieved through the sending and receiving of messages.

Debugging techniques for sequential programs rely heavily on the reproducible nature of such programs. If we repeatedly execute a given sequential program with the same given set of input data, we always obtain the same data and flow histories. However, this reproducible behavior cannot be guaranteed for concurrent programs, neither in a multiprocessor environment, where the processes execute on different processors at different speeds, nor in a single-processor environment, where the processor is switched among the processes, as a consequence of scheduling delays, the nondeterministic nature of process interactions, and lack of synchronization between the activities of the processes.

A possible approach to concurrent-program debugging is to consider each individual process in isolation and use sequential-program debugging techniques (e.g., controlled-execution techniques and tracing techniques) to discover errors within that process. However, the multiprocess composition of concurrent programs is, in itself, a potential source of a new classes of errors and, in particular, interprocess communication and synchronization errors.

Let us first consider controlled-execution techniques. In the debugging of a concurrent program, an essential feature of the trap-generating mechanism is the ability to generate a break trap (or breakpoint) on the occurrence of any interprocess interaction. Moreover, we must be allowed to restrict the trap to any subset of the set of processes that compose the program. However, even this capability is often not very useful because the act of inserting breakpoints may alter the overall behavior of a concurrent program. This is called the *probe effect.*

As far as the use of tracing techniques with concurrent programs is concerned, the problems connected with the memory space needed to keep the trace and the execution time required to gather the trace are compounded by the fact that we must record the activity of several processes. Keeping a copy of the whole program state and/or flow history may be impractical and is usually unnecessary; therefore, the use of some form of selective tracing is almost always mandatory. A possible approach considers the process as the unit of selective tracing, and records the activity of only a subset of the processes that constitute the concurrent program. In a different approach, one might collect information relevant to only a few aspects of the program activity (e.g., interprocess synchronization). When various processes of a concurrent program execute on different processors, it may not be entirely possible to figure out the exact order in which different events have taken place.

NONFUNCTIONAL DEBUGGING

Often the term *debugging* is used to denote the process of removal of bugs that may be affecting the functions or results computed by a program. However, there may be nonfunctional requirements associated with a program. For example, a program may be computing correct results, but its performance may be unacceptable according to its specification. Applications implemented using multiprocessors often encounter such problems. Therefore, one may need to fix the performance bug in this case. As another example, a real-time system may produce correct results but may not have acceptable response time. Similarly, a GUI (graphical user interface) may be found satisfactory from the viewpoints of its look and feel, ease of use, and so on.

CONCLUSION

Debugging is an unavoidable activity in software development, but it is often viewed as undesirable. Proper planning can ensure that debugging is not unnecessarily expensive or time-consuming. The use of appropriate tools and error classification schemes as aids to bug location can make debugging a relatively systematic process. In the limit, however, debugging is an intellectual exercise and one that software engineers must practice in order to gain skill and expertise.

Simple straightforward coding is a great help when debugging. It is easier to avoid and detect errors if the program is written in an orderly and logical manner. In the early stages of writing a complicated program, one should not hesitate to rewrite sections if doing so will simplify the program. Programming tricks should be avoided. The more tricks used when programming, the more difficult it is to debug one's own program. Tricky programs are nearly impossible to debug by someone who did not write the original program. This also touches on the subsequent maintenance and support of software. Recent estimates claim that the cost of maintenance amounts to 70% of the life cycle cost of a software product.

BIBLIOGRAPHY

1. M. Morcatty, *Software Implementation,* New York, Toronto: Prentice-Hall, 1991.
2. D. V. Tassel, *Program Style, Design, Efficiency, Debugging, and Testing,* Englewood Cliffs, NJ: Prentice-Hall, 1974.
3. G. J. Myers, *The Art of Software Testing,* New York: Wiley, 1979.
4. A. R. Brown and W. A. Sampson, *Program Debugging: The Prevention and Cure of Program Errors,* Amsterdam, The Netherlands: Elsevier, 1973.
5. Courant Computer Science Symposium, *Debugging Techniques in Large Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1970.
6. R. S. Pressman, *Software Engineering: A Practitioner's Approach,* New York: McGraw-Hill, 1988.
7. M. Bohl, *A Guide for Programmers,* Englewood Cliffs, NJ: Prentice-Hall, 1978.
8. J. B. Rosenburg, *How Debuggers Work: Algorithms, Data Structures, and Architecture,* New York: Wiley, 1996.
9. E Y. Shapiro, *Algorithmic Program Debugging,* Cambridge, MA: MIT Press, 1983.
10. B. Lazzerini and L. Lopriore, *Program Debugging Environments: Design and Utilization,* New York: Ellis Harwood, 1992.

LADAN TAHVILDARI
AJIT SINGH
University of Waterloo

SOFTWARE, COMPUTER COMMUNICATIONS.

See COMPUTER COMMUNICATIONS SOFTWARE.