Debugging and Testing

Meaning of Terms

Testing means verifying correct behavior. Testing can be done at all stages of module development: requirements analysis, interface design, algorithm design, implementation, and integration with other modules. In the following, attention will be directed at implementation testing. Implementation testing is not restricted to execution testing. An implementation can also be tested using correctness proofs, code tracing, and peer reviews, as described below.

Debugging is a cyclic activity involving execution testing and code correction. The testing that is done during debugging has a different aim than final module testing. Final module testing aims to demonstrate correctness, whereas testing during debugging is primarily aimed at locating errors. This difference has a significant effect on the choice of testing strategies.

Preconditions for Effective Debugging

In order to avoid excessive time spent on debugging, the programmer should be mentally prepared for the effort. The following steps are useful to prepare for debugging.

- **Understand the design and algorithm** - If you are working on a module and you do not understand its design or its algorithms, then debugging will be very difficult. If you don't understand the design then you can't test the module because you do not know what it is supposed to do. If you don't understand the algorithms then you will find it very difficult to locate the errors that are revealed by testing. A second reason for the importance of understanding algorithms is that you may need that understanding in order to construct good test cases. This is especially true for algorithms for complex data structures.
- **Check correctness** - There are several methods for checking correctness of an implementation prior to execution.
- **Correctness proofs** - One useful code check is to examine code using the logical methods of correctness proofs. For example, if you know preconditions, invariants, terminating conditions, and postconditions for a loop then there are some easy checks that you can make. Does the precondition, together with any loop entry code imply that the invariant is initially true? Does the loop body preserve the invariant? Does execution of the loop body make progress towards loop termination? Does the invariant,

together with the loop terminating condition and loop exit code, imply the postcondition? Even if these checks don't find all errors, you will often gain a better understanding of the algorithm by making the checks.

- **Code tracing** - Often, errors can be detected by tracing through the execution of various calls to module services, starting with a variety of initial conditions for the module. For poorly understood psychological reasons, tracing works best if you are describing your tracing to someone else. In order to be effective, tracing of a procedure or function should be done assuming that calls to other procedures and functions work correctly, even if they are recursive calls. If you trace into a called procedure or function then you will find yourself dealing with too many levels of abstraction. This usually leads to confusion. If there is any doubt about the called procedures and functions then they can be traced separately to verify that they perform according to specifications. Again, tracing may not catch all errors, but it can enhance your understanding of algorithms.
- **Peer reviews** - A peer review involves having a peer examine your code for errors. To be effective, the peer should either already be familiar with the algorithm, or should be given the algorithm and code in advance. When the reviewer meets with the code writer, the code writer should present the code with explanations of how it correctly implements the algorithm. If the reviewer doesn't understand or disagrees with part of the implementation, they discuss that part until both are in agreement about whether or not it is an error. The reviewer's role is only as an aid to detecting errors. It is left to the implementor to correct them. Much of the benefit of a peer review derives from the psychology of presenting how something works. Often the code writer discovers his or her own errors during the review. In any case, it is useful to have an outsider review your work in order to get a different perspective and to discover blind spots that seem to be inherent in evaluating your own work. Like code tracing, peer reviews can be time consuming. For class work, a peer review of an entire module is not likely to pay for itself in terms of instructional value. So reviews should be restricted to short segments of code. For commercial programming, however, quality of the code is much more important. Thus peer reviews are a significant part of a software quality assurance program.
- **Anticipate errors** - Unfortunately, humans make errors with correctness arguments and sometimes miss cases in code tracing, and peers don't always catch errors either. So a programmer should be prepared for some errors remaining in the code after the steps listed above. Hopefully, there won't be too many.

Requirements for Debugging

To effectively debug code you need two capabilities. First, you need to be able to efficiently call on the services provided by the module. Then you need to be able to get information back about results of the calls, changes in the internal state of the module, error conditions, and what the module was doing when an error occurred.

Driving the module

To effectively debug a module, it is necessary to have some method for calling upon the services provided by the module. There are two common methods for doing this.

- **Hardwired drivers** - A hardwired driver is a main program module that contains a fixed sequence of calls to the services provided by the module that is being tested. The sequence of calls can be modified by rewriting the driver code and recompiling it. For testing modules whose behavior is determined by a vary small number of cases, hardwired drivers offer the advantage of being easy to construct. If there are too many cases, though, they have the shortcoming that a considerable effort is involved in modifying the sequence of calls.
- **Command interpreters** - A command interpreter drives the module under test by reading input and interpreting it as commands to execute calls to module services. Command interpreters can be designed so that the commands can either be entered interactively or read from a file. Interactive command interpretation is often of great value in early stages of debugging, whereas batch mode usually is better for later stages of debugging and final testing. The primary disadvantage of command interpreters is the complexity of writing one, including the possibility that a lot of time can be spent debugging the interpreter code. This is mitigated by the fact that most of the difficult code is reusable, and can be easily adapted for testing different kinds of modules. For almost all data structure modules, the flexibility offered by command interpreters makes them a preferred choice.

Obtaining information about the module

Being able to control the sequence of calls to module services has little value unless you can also obtain information about the effects of those calls. If the services generate output then some information is available without any further effort. However, for many modules, including data structure modules, the primary effect of calls to services is a change in the internal state of the module. This leads to needs for three kinds of information for debugging.

- **Module state** - Data structure modules generally have services for inserting and deleting data. These services almost never generate output on their own, and often do not return any information through parameters. Therefore, in order to test or debug the module, the programmer must add code that provides information about changes in the internal module state. Usually, the programmer adds procedures that can display the data contents of the module. These procedures are made available to the driver module, but are usually removed or made private when testing is complete. For debugging, it is useful to have procedures that show internal structure as well as content.
- **Module errors** - When a module has a complex internal state, with incorrect code it is usually possible for invalid states to arise. Also, it is possible that private subroutines are called incorrectly. Both of these situations are module errors. When practical, code can be added to the module to detect these errors.
- **Execution state** - In order to locate the cause of module errors, it is necessary to know what services and private subroutines have been called when the error occurs. This is the execution state of the module. One common method for determining the execution state is the addition of debugging print statements that indicate entry and exit from segments of code.

Principles of Debugging

- **Report error conditions immediately** - Much debugging time is spent zeroing in on the cause of errors. The earlier an error is detected, the easier it is to find the cause. If an incorrect module state is detected as soon as it arises then the cause can often be determined with minimal effort. If it is not detected until the symptoms appear in the client interface then may be difficult to narrow down the list of possible causes.
- **Maximize useful information and ease of interpretation** - It is obvious that maximizing useful information is desirable, and that it should be easy to interpret. Ease of interpretation is important in data structures. Some module errors cannot easily be detected by adding code checks because they depend on the entire structure. Thus it is important to be able to display the structure in a form that can be easily scanned for correctness.
- **Minimize useless and distracting information** - Too much information can be as much of a handicap as too little. If you have to work with a printout that shows entry and exit from every procedure in a module then you will find it very difficult to find the first place where something went wrong. Ideally, module execution state reports should be issued only when an error has

occurred. As a general rule, debugging information that says "the problem is here" should be preferred in favor of reports that say "the problem is not here".

- **Avoid complex one-use testing code** - One reason why it is counterproductive to add module correctness checks for errors that involve the entire structure is that the code to do so can be quite complex. It is very discouraging to spend several hours debugging a problem, only to find that the error was in the debugging code, not the module under test. Complex testing code is only practical if the difficult parts of the code are reusable.

Debugging Aids

Aids built into programming language

- **Assert statements** - Some Pascal compilers and all C compilers that meet the ANSI standard have assert procedures. The assert procedure has a single parameter, which is a Boolean expression. When a call to assert is executed the expression is evaluated. If it evaluates to true then nothing happens. If it evaluates to false then the program terminates with an error message. The assert procedure can be used for detecting and reporting error conditions.
- **Tracebacks** - Many Pascal compilers generate code that results in tracebacks whenever a runtime error occurs. A traceback is a report of the sequence of subroutines that are currently active. Sometimes a traceback will also indicate line numbers in the active subroutines. If available, a traceback reveals where the runtime error occurred, but it is up to the programmer to determine where the cause lies.
- **General purpose debuggers** - Many computer systems or compilers come with debugging programs. For example, most UNIX operating systems have general purpose debuggers such as sdb and dbx. Debugging programs provide capabilities for stepping through a program line-by-line and running a program with breakpoints set by the user. When a line with a breakpoint is about to be executed the program is interrupted so that the user can examine or modify program data. Debugging programs also can provide tracebacks in case of run-time errors. Debuggers are often difficult to learn to use effectively. If they are the only tool used for debugging then it is likely that they will not save much time. For example, debugging a data structure module with a debugger, but without a good test driver, will likely result in spending a lot of time getting piecemeal information about errors.

Debugging Techniques

Incremental testing

In a good design for a complex module, the code is broken up into numerous subroutines, most of which are no more than 10 to 15 lines long. For a module designed in this way, incremental testing offers significant advantages. For incremental testing, the subroutines are classified in levels, with the lowest level subroutines being those that do not call other subroutines. If subroutine A calls subroutine B then A is a higher level subroutine than B. The incremental testing strategy is to test the subroutines individually, working from the lowest level to higher levels. To do testing at the lower levels, the test driver must either be capable of calling the low level subroutines directly, or else the programmer must be able to provide several test input cases, each of which only involves a small number of low level subroutines. Devising these test cases requires a thorough understanding of the module algorithms, along with a good imagination. The strength of incremental testing is that at any time in the process, there are only a small number of places where errors can arise. This automatically makes debugging information more meaningful and leads to quicker determination of the cause of an error. A second reason for incremental testing is that it greatly reduces the chances of having to deal with two or more errors at the same time. Multiple errors often will generate confusing error indications.

Sanity checks

Low level code in complex data structure is often written with the assumption that the higher level code correctly implements the desired algorithm. For example, the low level code may be written with the assumption that a certain variable or parameter cannot be NULL. Even if that assumption is justified by the algorithm, it may still be a good idea to put in a test to see if the condition is satisfied because the higher level code may be implemented incorrectly. This kind of check is called a sanity check. If an assert procedure is available then it can be used for the checks. The advantage of sanity checks is that they give early detection of errors.

Boolean constants for turning debugging code on or off

If debugging code is added to a module then it is often profitable to enclose it in an if statement that is controlled by a Boolean constant added to the module. By doing this, the debugging code can easily be turned off, yet be readily available if needed later. Different constants should be used for different stages of testing so that useless information is minimized.

Error variables for controlling program behavior after errors

When debugging print statements are added to code, there is the possibility of a tremendous explosion of useless information. The problem is that a print statement by itself will be executed whether or not there is an error. Thus, if the error does not appear until a large number of subroutine calls have been made then most of the messages are just telling you everything is okay so far. This problem is greatly magnified if the added code is displaying the internal structure of a data structure. Assuming that the module has sanity checks for error detection, an error Boolean variable can be added to the module. It should be initialized to false, indicating that there is no error. For most data structures, there is a Create operation for initialization. The error variable can be initialized at the same time. Instead of exiting the sanity checks are modified so that they set the error variable to true. Then debug code can be enclosed in if statements so that information is only printed when errors have been detected. One possible application of this method is obtaining traceback information when it is not otherwise available.

Traceback techniques

To obtain a traceback, use an error Boolean set by sanity checks. At various places in the module add debug code controlled by the error variable that prints the current position. Usually it is more economical to first run the code with a terminating sanity check. Then you only need to add the controlled debug code at places where the subroutine that contains the sanity check is called.

Correcting Code Errors

For the correction of errors detected by testing, the is one very important principle to keep in mind: **fix the cause, not the symptom**.

Suppose that you run some code and get a segmentation fault. After some checking you determine that a NULL pointer was passed into a procedure that did not check for NULL, but tried to reference through the pointer anyway. Should you add a NULL pointer check to the procedure, enclosing the entire body of the procedure in an if statement? This question cannot be answered without an understanding of the design and algorithm. It may be that if the algorithm is correctly implemented then the pointer cannot be NULL, so the procedure does not make the check. If that is the case then adding the if statement does not fix the cause of the problem. Instead, it makes matters worse by covering up the symptoms. The problem will surely appear somewhere else, but now the symptoms will be further removed from the cause. Code such as the pointer NULL check should be added only if you are sure that it should be part of the algorithm. If you add a NULL pointer check that is not required by the algorithm then it should report an error condition. In other words, it should be a sanity check. [an error occurred while processing this directive]