Palo Alto Research Center

An Effective Test Strategy

Howard Sturgis

XEROX

An Effective Test Strategy

Howard Sturgis

CSL-85-8 November 1985 [P85-00027]

Abstract: In this paper I describe a debugging strategy that I have successfully used for several years. The principal idea is to excise test subjects from large programs and test them in individually crafted test beds, where the test beds are constructed according to four principles: (1) provide an "encapsulation" of the test subject that presents deterministic behavior to the test driver, (2) write a program to simulate the behavior of the encapsulated test subject, (3) use a random-number generator to construct the test sequences and compare the resulting behavior of the encapsulated subject with that of the simulation, and (4) provide the ability to exactly repeat a test sequence so as to isolate a detected bug through "binary chop."

While this strategy does not have the theoretical completeness provided by verification, I have found it to be easy to implement and considerably more effective than haphazard debugging. In all cases where I have used it, it has required much less time to construct the test beds than required for the original design and implementation of the program being tested. Further, I generally find that the resulting program is about as bug-free as it would have been had formal verification been available and used.

CR Categories and Subject Descriptors: D.2.5 **[Software Engineering]:** Testing and Debugging.

Additional Keywords and Phrases: testing, debugging aids, test data generator, programming research.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Introduction

In this paper I describe a debugging strategy that I have successfully used for several years. The principal idea is to excise test subjects from large programs and test them in individually crafted test beds, where the test beds are constructed according to four principles:

1. provide an "encapsulation" of the test subject that presents deterministic behavior to the test driver,
2. write a program to simulate the behavior of the encapsulated test subject,
3. use a random-number generator to construct the test sequences and compare the resulting behavior of the encapsulated subject with that of the simulation, and
4. provide the ability to exactly repeat a test sequence so as to isolate a detected bug through "binary chop."

While this strategy does not have the theoretical completeness provided by verification, I have found it to be easy to implement and considerably more effective than haphazard debugging. In all cases where I have used it, it has required much less time to construct the test beds than required for the original design and implementation of the program being tested. Further, I generally find that the resulting program is about as bug-free as it would have been had formal verification been available and used.

This strategy was developed in the Xerox Parc Computer Science Laboratory over a period of several years. It has been applied to a number of programs, including a prototype file server, a B-tree package, microcode to implement a floating point package, and a microcode implementation of a reference count package supporting a garbage collecting allocator [Rovner84].

During this same period, the Computer Science Laboratory has developed a succession of symbolic debugging environments, such as that provided by the Cedar operating system [Swinehart85] [Teitelman84]. These debugging environments have made the analysis of program bugs particularly easy, generally requiring small numbers of minutes to determine the circumstances of a particular bug. As a consequence, most of our programs are debugged *in vivo*.

Debugging *in vivo* has two well known limitations. First, many program errors are provoked infrequently, and may not show up during testing, or may show up only after seemingly minor changes to the program environment. Second, a program error may not be detected until long after the actual failure occurred, so that sufficient evidence may not remain to diagnose the fault; further, due to an asynchronous environment, it may not be possible to reliably repeat the error for analysis.

A method frequently proposed for reducing these limitations is to break a large program up into components, and test each component in an individually tailored test bed, i.e., *in vitro*. These test beds can be designed to provoke otherwise infrequent bugs, and to compare the behavior of the program with independently defined expected behavior.

The strategy described in this paper is one method for constructing these test beds. It was developed by trial and error, as I attempted to apply the above advice. Rather than describe the

individual problems, I describe the strategy as a whole, together with a rationale for its components. The first part of the paper is devoted to that description. The second part of the paper describes the application of this strategy to a particular example: 700 lines of microcode implementing a set of reference counting op-codes. This example is illustrative of my experiences with this strategy, both in terms of ease of application and results.

The Strategy

Debugging

The strategy should be considered in light of the typical debugging cycle. Finding and fixing a single bug has three conceptual steps:

1. provoking the bug,
2. detecting the resulting erroneous behavior, and
3. isolating the detected bug.

The four test bed design principles described above support these three steps in various ways. The encapsulation and randomly generated test sequences bear the responsibility for provoking bugs. A correct simulation supports detection of the bug through an eventual difference in the behavior of the simulation and the encapsulated subject. An ability to exactly repeat a failed test sequence supports the isolation of the detected bug through binary chop.

Test Subjects

Before getting into the details of the test beds, we consider some possible test subjects that may be encountered. These vary from simple to complex.

The simplest form of test subject is a pure function, e.g., a floating point function. This is a single procedure, taking one or more floating point arguments, and returning a single result; its behavior on each call is completely independent from previous calls, depending only on the arguments to the particular call, and calls on the square root routine affect no other system activity.

A slightly more complicated form of test subject is a package of one or more procedures that together manage some piece of internal state; an example is the ever-popular "stack." The internal state of a stack is a sequence of values, and the procedures include Push and Pop. The behavior of a single procedure call on one of these test subjects is dependent only on the internal state value and the actual parameters to the procedure call. That is, the returned values and the final internal state value are functionally determined by the initial internal state value and the actual parameters to the procedure call. In other words, a call on a package procedure performs an atomic update to the

internal state. Such packages are frequently called abstract datatypes [Shaw84].

One could treat these abstract datatype test subjects as examples of the previous case, where the current value of the stack is passed as one of the arguments, and the new value is one of the results. However, because the internal state may get very complicated and require large amounts of storage for representation, programming systems frequently provide special mechanisms for representing the internal value. An extreme case is a file system implemented on a disk; the internal state representation includes the physical contents of the disk.

The above two classes of test subject can be characterized as interacting with their external environments purely through the initial procedure call and the subsequent return, so long as their internal state is not considered as part of their environment. Thus their behavior is totally determined by their initial internal state and actual parameters; any activity concurrent with their execution cannot affect their behavior. However, there are examples of test subjects that do not satisfy these conditions.

A simple example of such a subject is a package implementing some form of remote procedure call (RPC) [Birrell84], to be tested in isolation from the underlying primitive communication machinery. In normal usage, a single call on the RPC package is a succession of events: (1) the initial procedure call from the client code to the RPC package (divorced from the subsequent return of this procedure call), (2) an exchange of messages between the RPC package and a remote computer, and (3) the return of the initial procedure call to the client. Figure 1 shows the RPC package in normal use. The returned values depend on the specific interchange of messages and, hence, on the initial state of the remote computer, as well as chance events in the underlying communication mechanism. One could treat the triad (RPC package, communication mechanism, and remote computer) as a single entity. In this case, one might obtain behavior much like that described above for abstract datatypes. However, if one wants to test the RPC implementation independently, there is more to consider.

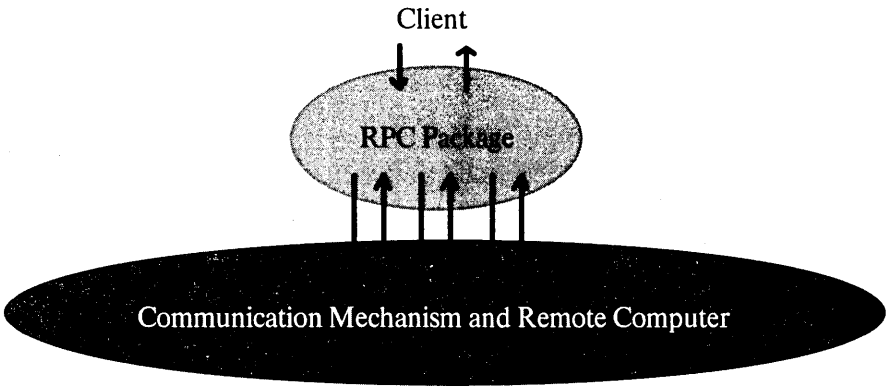

Figure 1. A test subject in normal use.

One way to think about such a package is to draw an imaginary line around it, and ask what interactions cross that line and affect either the package's subsequent behavior or the subsequent

behavior of the environment. If we perform this exercise on the stack example, we see just the initial procedure call and the subsequent return. When we do this for the RPC package, we see the initial procedure call and the subsequent return; but we also see calls on the underlying communication mechanism, their subsequent return, as well as possible error indications. Other test subjects may interact with their environment by reading and writing global data structures accessible by other concurrent programs. The microcode to be described below contains this last sort of interaction.

Once we have drawn a line around the package and determined the significant interactions with the environment, then we can describe the package as having an internal state. In each internal state, the package is permitted to have only certain interactions with the environment. Each of these interactions is "atomic," resulting in some information transfer into and out of the subject, followed by a new internal state.

For example, the RPC package may have an idle state in which the allowed interactions are incoming procedure calls requesting a remote interaction. The procedure call results in a new state in which the only allowed interaction is the sending of a message to the remote machine. (The representation of this internal state includes the "frame" of the called RPC implementation procedure.) Sending this message results in an internal state in which only two interactions are possible, the receipt of a message from the remote machine or a timer interrupt. After receipt of a message from the remote machine, the next state will contain information from the received message, and will only allow a single interaction, the final return to the original caller. Following this final return, the package is in its original idle state. A timer interrupt is likewise followed by a single interaction, the final return to the original caller, indicating an error. (Of course, real RPC packages allow for far more interactions.)

Thus, we have three distinct varieties of test subject: pure functions, abstract datatypes, and complex packages.

Encapsulation

Encapsulation isolates a test subject from extraneous influences, provides a clean procedural interface to be called by the test driver, and provides for additional access by the test driver to its internal state. Isolating a complex package makes it behave as if it were an abstract datatype. We now consider the three varieties of test subject.

We begin with pure functions, as represented by floating point procedures. It would seem that these are already in the appropriate form, so that the test driver need only generate a sequence of floating point numbers, call the floating point procedure with each one, and compare the actual result with the expected result. Sometimes, however, the procedure may not return in a normal manner. For example, when presented with a negative argument, a square root routine may "trap," or invoke some other "exception" mechanism. The purpose of this trap is to relieve the client from having to check an error flag after each call.

In this situation, we can design the encapsulation to "catch" the exception, and convert the

exception into an ordinary return to the caller. Such an encapsulation supplies a procedure to the test program. This procedure accepts a floating point number as an argument and returns a two element record: a Boolean flag to indicate whether a trap occurred and a floating point result. With this encapsulation of the square root routine, the test program can always expect an ordinary procedure return, and can test the resulting flag and value as appropriate to the arguments of the call.

An encapsulation may hide details that are not being tested. For example, we may only be interested in learning if the floating point routine can "crash" the system. (This would be interesting if the floating point routine were implemented in microcode.) In this case, the entry procedure in the encapsulation need not return any results at all, an error is indicated by the crash of the system.

Encapsulation for abstract datatypes is almost the same as for pure functions; the only difference is that it is frequently useful to add procedures to inspect the internal state of the subject. An initialization procedure may also be needed. Inspection procedures need not return specific information, but may simply perform a consistency check. During a binary search to isolate a bug, these inspection procedures can be invoked between successive tests to look for failures that have not as yet affected external behavior. One can also perform minor surgery on the test subject to initiate calls on the check procedures from time to time during complex operations, further isolating a bug.

Complex test subjects require encapsulation in order to attain reproducible behavior. Encapsulation supplies a simulated external environment for the test subject. The goal is for the test subject together with its simulated environment to behave just like an abstract datatype. That is, each procedure call from the test driver causes a deterministic change in the combined internal states of the test subject and the encapsulation, followed by a procedure return to the test driver. However, arranging for abstract datatype behavior is not enough; the simulated environment must provide sufficiently varied behavior so as to provoke latent bugs in the subject.

Considering the previously described RPC package will make this clear. Assume that the RPC package transmits messages by calling a procedure normally supplied by the operating system. As part of the simulated environment, the encapsulation provides a replacement "transmit" procedure. When this replacement procedure is called, the encapsulation can decide whether to return with a normal response, or generate an exception representing a time out. The encapsulation also supplies a procedure to be called by the test driver to initiate a single test. Parameters to this call control the response by the substitute procedure to transmission requests from the RPC package. Upon being called by the test driver, the encapsulation calls the RPC package with appropriate parameters. Figure 2 shows the RPC package encapsulated for testing purposes.

A test subject may involve multiple concurrent processes interacting through some data private to the subject. In this case, the encapsulation must control the relative order of execution of the individual processes whenever there is more then one ready to run at a time (the process scheduler is part of the simulated environment). Further, the encapsulation must be prepared to shift execution from one process to another following each atomic interaction between a process and the subject's private data. A similar situation arises when the test subject is a collection of programs running on

different machines, using some form of asynchronous communication. In this case, the encapsulation must somehow control the order in which messages arrive at the individual machines, and the occurrence of lost messages and repeated messages, so as to simulate asynchronous behavior of the machines in a reproducible way. This form of encapsulation converts each client call on the test subject into a single atomic act acting upon the combined internal state of the subject and the encapsulation, which is a succession of individual interactions between the subject and the encapsulation. The microcode package described below is an example of one of these complex subjects.

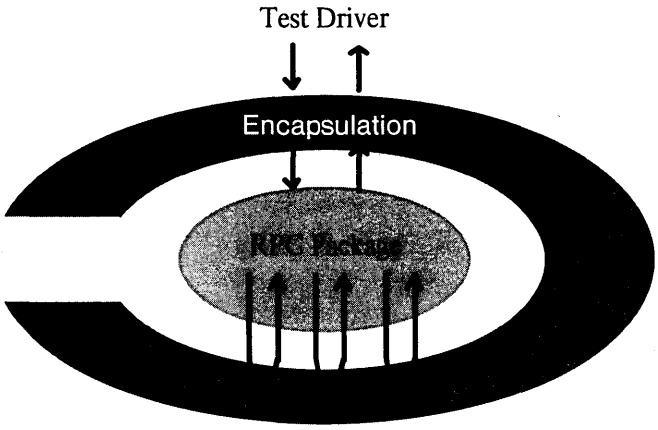

Figure 2. An encapsulated test subject.

There is another form of encapsulation which I have not tried—to convert each interaction with the simulated environment into a single atomic action of the combined subject and encapsulation. In this form, whenever the test subject would initiate an interaction with its environment, its current state would be frozen, and the encapsulated subject would make a procedure return to the test driver. Such an encapsulation would require a sophisticated operating system environment (concurrent processes and monitors, for example). Further, if the test subject interacts with its environment by means of reads and writes of global data (in addition to procedure calls), then surgery on the test subject would be required to freeze it before each read and after each write. This would enable the encapsulation to gain control and make the required procedure return to the test driver.

In brief, the purpose of encapsulation is (1) to provide a simple procedural interface, including inspection procedures providing access to the internal state of subjects, (2) to convert complex subjects into apparent abstract datatypes, and (3) to provide a sufficiently rich environment for the complex subjects so as to provoke latent bugs.

Simulation

The primary purpose of simulation is to exactly mimic the desired behavior of the encapsulated test subject. In addition, the simulation can provide useful information for the test generator.

Two things are required to detect a bug in the test subject. One is to provoke it, which is the responsibility of the test generator and the encapsulation, and the other is to detect it, which is the responsibility of the simulation which needs to provide detectably different behavior. Once this difference in behavior is detected, then it is the job of the human programmer to track down the ultimate cause: a fault in the test subject, encapsulation, or simulation.

During testing, we expect to see a series of detected errors, each resulting in a correction to the test subject, the encapsulation, or the simulation. Ultimately, no differences in behavior will be detected.

Of course, an initial attempt to write a simulation will probably contain errors, i.e., it won't describe exactly what is desired from the test subject. This happens because the simulation is just another program, and programs contain bugs. However, the situation is not as bad as it seems, since most bugs in the simulation are unlikely to mask bugs in the test subject. All that is required is that whenever there is a bug in the encapsulated subject, the simulation should behave differently from the encapsulated subject.

If the simulation misbehaves in the same way as the test subject, this is usually due to an incorrect informal specification, correctly implemented in both the test subject and the simulation. On the other hand, I have only rarely experienced situations where a correct specification was incorrectly implemented, in both the simulation and the test subject, leading to the same incorrect behavior.

Usually a simulation is much simpler to write than the actual test subject, as there is little need to use sophisticated algorithms to attain efficiency. Further, a complex test subject may be needed to handle asynchronous environments, but we need only simulate the deterministic behavior of the encapsulated test subject (of course, some of the detection responsibility has now moved to the design and implementation of the encapsulation). Being simpler to write, and using different algorithms, leads to different errors of implementation.

In addition to providing a standard of comparison for the test subject, the simulation can provide useful information for test generation. For example, if the test subject is a symbol table, then the simulation can provide procedures to select a suitable symbol for a subsequent test. This might be a symbol known to be in the table, or one known to have once been in the table and subsequently deleted.

Test Generation

The task of test generation is to provoke any errors in the test subject; for complex test subjects, this responsibility is shared with the encapsulation. There have been many proposals in the literature for generating comprehensive test sequences, or testing the completeness of a specific test sequence; e.g., see [Budd78] and [Goodenough75]. These tend to be difficult to implement, require a fairly rigorous understanding of the test subject, and frequently require an implementation specific to the programming language. Instead, I have had good success from generating test sequences using a pseudorandom-number generator applied with "seat of the pants" reasoning. Of course, these tests are

not necessarily complete. In the microcode example I discuss later, I observe that the number of faults remaining due to incomplete testing was roughly comparable to the number remaining due to incorrect specification; thus there would have been little practical benefit from an exhaustive test. It is very important, however, to apply the pseudorandom-number sequences judiciously.

For test subjects that contain no internal state, i.e., functions, test construction is relatively easy. One simple method is to generate arguments from a uniform distribution over the argument space. A short examination of the function implementation usually leads to a considerable improvement over this simple sequence.

As an example, consider the following generator for a test sequence for a floating point add routine. First, choose whether to generate the two arguments independently or not. Next, choose for each field in an argument (sign, exponent, fraction), a density of one bits: high, medium, or low. Then, generate the field, bit by bit using an appropriate probability for a one bit. Finally, if the two arguments are to be related, construct a random bit pattern to xor with one argument to produce the other. The point here is that this is not a rigorous design intended to test each possible path through the add routine, but rather these possibilities were selected from a knowledge of the sorts of errors one might encounter in a floating point package. Further, we needn't do an exhaustive analysis of the routine to find exactly the right test arguments. We can rely on the random-number sequence to eventually produce the right ones, given that we have directed the exploration into the right area, e.g., lots of one bits in the exponent frequently produces large exponents.

Test subjects that contain internal state (abstract datatypes) are more difficult. Here it is not only necessary to provide the arguments that provoke a bug, but also to drive the internal state to appropriate values. This may require designing the test to run in phases, driving the internal state to different regions during the different phases. For example, consider a B-Tree implementation [Bayer72]. It is well known that B-Tree implementations are prone to errors when increasing or decreasing the depth of the tree. This suggests a test design with two phases. In one phase, there are more additions than deletions, so that the tree gradually grows. In the other phase, there are more deletions than additions, so that the tree gradually shrinks. By supplying appropriate inspection routines, either in the encapsulation or in the simulation, the test generator can choose when to switch from one phase to the other.

Complex test subjects add little difficulty to test generation, except noting that part of the responsibility for provoking bugs now resides in the encapsulation. For example, when testing a communications package, the encapsulation will present the behavior of a communications channel. As such, it must provide all of the possible behavior in such a channel: lost messages, garbled messages, and messages arriving out of order. A simple way to provide this behavior is to allow the encapsulation to call on the random-number generator when deciding how to respond to a particular request from the test subject.

Test Control

The overall meta-program for testing is to search for discrepancies in the behavior of the encapsulated subject and the simulation, isolate the discrepancy to its "root" cause, repair the fault, and then begin searching anew. Since it is at best difficult, and usually impossible, to discover the cause of a discrepancy from the observed behavior, it is essential to be able to repeat exactly the sequence of test stimuli that led to the discrepancy. This permits a "binary chop" approach to isolating the fault.

Two factors in this test design permit exact repetition of a test sequence: designing the encapsulation so that the encapsulated subject behaves deterministically, including its succession of internal states, and using a pseudorandom-number generator, started from a known "seed," to generate test sequences.

In addition, since some subjects with complicated internal state may require a long period of time to randomly walk into "interesting" corners of their state space, facilities are needed to save intermediate states, and to replay from such saved states.

Summary of important points

1. The goal of testing is to provoke and detect all errors in the subject. The goal of debugging is to localize and repair the errors.

2. The behavior of the simulation must differ from that of the subject whenever the subject is incorrect; the simulation need not be completely free of bugs.

3. We "encapsulate" the subject program to provide a "cleaner" interface to be emulated by the simulation, and to provide repeatable behavior.

4. Test sequences are generated from pseudorandom-numbers. Their distribution must be chosen with the knowledge of the subject's implementation.

5. The tests must be exactly reproducible, so as to allow fault isolation by binary chop.

A Case Study of Reference Counting Op-Codes in Cedar

As an example of the application of this testing strategy, I describe my experiences in testing and debugging a set of microcoded op-codes for the Cedar programming language, a research prototype language in use in the Computer Science Laboratory at Xerox PARC.

The Project

The Cedar language is an extension of the Mesa programming language [Mitchell79]. One of the extensions is an automatic garbage collected storage allocator [Rovner84], based on a reference counting mechanism. This mechanism is implemented through a suite of special op-codes that

implement the counts and inspect them during garbage collection.

At the beginning of this project, version 4.0 of the Cedar language had already been implemented on two different processors, the Dorado [Xerox 1132] and the Dolphin [Xerox 1100]; and it was desired to provide an implementation for the Dandelion [Xerox 1108]. On each of these processors, the language is provided through a collection of pseudo-op-codes, implemented by an interpreter written in the microprocessor language of the specific machine. As there was already an existing implementation of the Mesa language on the Dandelion, we needed an implementation of the additional op-codes required for Cedar 4.0, including the 11 reference counting op-codes.

When I began the project, there was no formal description of these op-codes; instead I was offered the existing implementations on the Dorado and the Dolphin, together with a prose description of the op-codes intended to guide the design of a new processor. Because of its clarity, I chose to use the prose description as the single basis of my implementation.

Cedar Reference Counting Op-Codes

The eleven Cedar reference counting op-codes, hereafter referred to as "ref-counting" op-codes, support the counting mechanism and are invoked when references are created and destroyed, as well as during a garbage collection. The reference counts are not maintained in storage directly associated with the allocated record, but rather in a chained hash table, whose keys are the addresses of the allocated records. Not all allocated records are in this table, as those with reference counts equal to one are not included. The assumption is that most allocated records have a reference count of one, so that this table records only the exceptions.

These op-codes are an example of a complex test subject. First, the ref-counting op-codes are implemented in microcode as part of the op-code interpreter, and thus manipulate the representation of the Cedar internal machine state (e.g., local and global frame pointers and the expression evaluation stack). Errors in the implementation of these op-codes have the capacity to destroy the Cedar machine abstraction, upon which the test code is itself running. These effects can be quite subtle, and may not show up for a long time after the op-code has completed execution. As an example, if an op-code leaves an extra value on the interpreter evaluation stack, this may cause a subsequent interpreted Monitor Entry op-code to behave erratically, without affecting the behavior of any intermediate op-code.

Second, these ref-counting op-codes manipulate a data structure holding the global reference counts, as well as the representation of the Cedar internal machine state. This data structure is represented in the address space of the running Cedar program, and is also manipulated by system software using non ref-counting op-codes.

Third, some ref-counting op-codes are not defined as atomic actions, but rather are defined as executing in successive stages. These stages manipulate all components of the represented machine state, including the expression evaluation stack and the reference counts. During normal operation, a multiple stage op-code may complete one or more stages and then interrupt execution. These

interruptions can be due to page faults or higher priority process interrupts. While interrupted, op-codes from other processes are allowed to execute, including ref-counting op-codes which may themselves modify the reference counts. Following the interruption, the op-code is restarted beginning with its first stage. This loop may occur several times before the op-code finally completes and execution moves to the next op-code in the program sequence.

It is the responsibility of the implementor of the op-code to be sure that interruptions only occur between stages; a typical implementation error is to encounter a page fault after some global changes have been made, but before all of the changes defined for a particular stage.

Fourth, not all of the components of the processor state that affect or are affected by the operation are accessible to the test code. For example, page faults and higher priority process interrupts. Thus, successive invocations of the same op-code may result in different behavior.

Finally, these ref-counting op-codes constitute part of the Cedar machine abstraction on which the test code runs. The test code will itself invoke these op-codes. This is a difficulty, both because the op-codes initially might have bugs, and because their execution will disturb the reference counts on which the subject op-codes depend.

Ref-Counting Op-Codes Encapsulation

I was unable to completely encapsulate the ref-counting op-codes following the principles described above because they are embedded in an already existing implementation and manipulate components of underlying machine-state. Furthermore, when exceptional conditions occur, a ref-counting op-code may jump directly into existing code. Also, the ref-counting op-codes are being called as part of normal Cedar execution. This inability to completely encapsulate is a common occurence when debugging complicated code. In view of this incomplete encapsulation, I took a number of additional precautions.

I constructed two sets of entry points for the implementation, assigning distinct op-code numbers to each. This way, the test package could explicitly call on individual ref-counting op-codes, while they were also being called as part of normal Cedar execution. I allocated a private instance of the reference counts. When entered at the test entry points, the microcode would use the private instance rather than the global instance. Once called, the subject microcode was allowed to run until it either interrupted or completed. If it interrupted, it was repeatedly restarted in the normal manner until it eventually completed (of course, other processes ran in between the restarts).

The ref-counting op-codes may terminate by calling exception handlers in the non-microcode portion of the ref-counting package. For the test entry points, I replaced these with calls on exception handlers embedded in the encapsulation.

My encapsulation provided procedures for inspecting and modifying the private copy of the reference counts. I included procedures for forcing the "swap out" of pages of the reference count data structure. I could therefore consistently force the ref-counting op-code implementations to interrupt execution due to page faults even though I could not consistently prevent them from interrupting

execution. I also installed consistency checking microcode that could detect some instances of an interruption in an unclean state.

The resulting encapsulation produced code that was repeatable so long as it encountered no faults. Even then, it was repeatable unless the fault involved an interruption in an "unclean state" that was not a result of my forcing out a page of the reference count data structure. I hoped that such faults would eventually occur as a result of my forcing out a page, and would therefore be repeatable for isolation by "binary chop."

The existing Cedar implementation had provisions for using software implementation rather than microcode implementation of the ref-counting op-codes. During initial debugging, I used this software implementation rather than my microcode implementation.

Debugging was greatly facilitated by a symbolic microcode debugger that allowed planting breakpoints and inspection of the internal microcode registers. So, even though a bug could catastrophically destroy the microcode state, and many did, it was possible to replay the events leading up to the destruction.

A History of Debugging the Cedar Ref-Counting Op-Codes

It required about 700 Dandelion micro instructions, written over a period of three months, to implement the ref-counting op-codes. Debugging proceeded in two periods. During the first period—4 weeks—I debugged using only the test environment, while during the second period—2 weeks—I used a combination of the test environment and live execution in the Cedar system. During the first period I detected and repaired 38 microcode bugs, along with a similar number of bugs in the test code. During the second period I detected and repaired five microcode bugs.

Following the completion of the second debugging period, the microcode was used for six months in several personal workstations, including mine. No bugs were noticed involving this microcode. At the end of the six month period, the microcode was replaced by an entirely new implementation of the Cedar garbage collector, using different algorithms and data structures.

During the first four weeks, I combined debugging activities with programming the test environment. I would frequently write a simulation of an op-code, only to discover that I did not completely understand its intended behavior. These situations would generally involve rare events. For example, exactly how do we test for the end of the chain of free overflow cells? There were less than ten microcode errors discovered this way, and I did not count any of these among the 38 errors mentioned above.

Individual bugs were generally quite easy to understand and repair—in most cases, a half hour per bug. One exceptional bug took an hour to track down; it was the only bug that manifested itself as random behavior during testing. (The implementation was incorrectly cleaning up after itself, leaving the microcode registers in an inconsistent state.) During this period, my records show that I fixed an average of two microcode bugs a day, wrote test code, and repaired about two microcode bugs a day in this test code. I actually worked in two modes. In one mode I wrote new test code, and in the other I

used that code to find and fix bugs. When in the bug-finding and fixing mode, I detected and repaired five to ten bugs per day.

On the final run of this first period, the test ran 22 hours, performing 150,000 individual tests, with 1000 tests between each re-initialization of the global state. In all test runs, no bug occurred after the first 4000 tests. Most bugs were detected within the first 200 tests of a run. One op-code was not under test because at the time I did not see how to test it without destroying the test environment. It turned out that I did not correctly understand how it was supposed to function.

As I could not think of any obvious way to increase the power of the test, and since I had become very curious, I decided to run my microcode implementation in a live Cedar system. During the next two weeks I tracked down and repaired five additional microcode bugs (a rate less than the previous two per day). These bugs were more difficult to fix because they were initially only observable in the live system, and were generally not repeatable. I was eventually able to reproduce each of them in the test environment, where they behaved in a repeatable fashion. Of these five bugs, two could have been discovered by the original test, but had not been provoked, and three could not have been discovered, because the same misconceptions were programmed into the simulation as were in the subject microcode.

The two bugs that were not provoked occurred in a section of microcode that was not even entered during the initial testing. If the microcode debugging environment had some mechanism for determining branches of microcode that had been executed, then these bugs would not have survived the initial testing.

Discussion

It is frequently argued that because testing can only show the presence of bugs, not their absence, that one should concentrate on providing full verification, rather than constructing fancy debugging environments. Consider how verification would have fared for this microcode.

First, we must assume that whoever writes the formal specification for the microcode has available no more information than I did. Hence, the specification would probably include the same misconceptions as were included in my simulation. (In fact, two of the three misconceptions were clearly included in the original prose document.) Thus, while the microcode contained five bugs after testing, it would have contained three after full verification. In this case, the result of testing (five bugs) is little worse than the probable result of verification (three bugs).

The cost of constructing this test bed was relatively small. The total debugging time for these 700 lines of microcode was 6 weeks, less than the 12 weeks spent writing them. Of this 6 weeks, at most two were spent implementing the test environment, including the encapsulation, simulation and test generation.

Unfortunately, the encapsulation did not provide complete isolation for the test subject. However, this affected only one bug detected running *in vitro*, all the rest behaved in a deterministic fashion. Thus, even though this was a fairly complex test subject, it was effectively isolated.

Conclusion

This paper has described a strategy that can be implemented by an ordinary programmer. There are two major advantages that this strategy gains over normal debugging practices: (1) its power is roughly comparable to that of verification, while considerably easier to implement; and (2) once a bug has been detected, repeatability makes it easy to find and repair.

The case study of reference counting op-codes in Cedar supports these points. A complete verification would have discovered 40 errors (plus perhaps some additional bugs that caused no difficulty in the installed system) and would probably have missed the 3 additional errors that could be attributed to inadequate specification. The actual testing found 38 of the 40. Further, it took 12 weeks to write the microcode and 6 weeks to debug it. It is unlikely that verification could have been performed in the 6 weeks required to debug the code, assuming the technology even existed to verify this kind of microcode.

A comparison of my experiences with the first 38 bugs and the subsequent 5 bugs supports my second point. None of the 38 bugs took long to isolate once they had been detected; the most difficult taking one hour. The next 5 bugs took two weeks to isolate, even though they were "detected" instantly; i.e., the system crashed as soon as the microcode was installed. One might argue that these bugs were inherently difficult to isolate; that is, even if the test had provoked and detected them, they still would have required a long time to isolate. However, after examining each of them, I believe that I could have localized them as quickly as I localized the other 38; they had nothing to particularly distinguish them from the other bugs.

Acknowledgments

Two former members of the Xerox Palo Research Center staff contributed to the development of the techniques described in this paper. Jim Mitchell and Jim Morris were fellow members of a File Server project, and constructed several test packages for components of the server. Jim Morris, in particular, supplied the initial spark for the development of these ideas; after I had spent several days constructing a complete test for a certain file server interface, he showed me an equally effective test based on random-number generation that he had constructed in a much shorter period. Finally, I thank Robert Ritchie, who supplied needed encouragement while reading several drafts of this paper.

References

[Bayer72] R. Bayer and E. McCreight. *Organization and Maintenance of Large Ordered Indexes.* Acta Informatica 1, 1972, pp. 173-189.

[Birrell84] A. Birrell and B. Nelson. *Implementing Remote Procedure Calls.* ACM TOCS (2)1, Febuary 1984.

[Budd78] T. Budd, R. Lipton, F. Sayward, and R. DeMillo. *The Design of a Prototype Mutation*

System for Program Testing. AFIPS 1978 National Computer Conf., Anaheim, CA, June 5-8, 1978; pp. 623-628.

[Goodenough75] J. Goodenough and S. Gerhart. *Toward a theory of test data selection.* IEEE Trans. Softw. Eng. SE 1-3, June 1975, pp. 156-173.

[Mitchell79] J. Mitchell, W. Maybury, and R. Sweet. *The Mesa Langauge Manual.* Xerox PARC Technical Report CSL-79-3; 1979.

[Rovner84] P. Rovner. *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.* Xerox PARC Technical Report CSL-84-7, 1984.

[Shaw84] M. Shaw. *Abstraction Techniques in Modern Programming Languages.* IEEE Software, October 1984, pp. 10-26.

[Swinehart85] D. Swinehart, P. Zellweger, and R. Hagmann. *The Structure Of Cedar.* ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments. Seattle, Washington, June 25-28, 1985, pp. 230-244.

[Teitelman84] W. Teitelman. *A Tour Through Cedar.* IEEE Software, April 1984, pp. 44-73.

XEROX

An Effective Test Strategy

Howard Sturgis

CSL-85-8