SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 51–63 (1979)

Debugging Techniques

SØREN LAUESEN
Institute of Datalogy, University of Copenhagen

SUMMARY

Debugging is efficient if it detects all program errors in a short time. This paper discusses several techniques for improving debugging efficiency. Attention is given both to the initial debugging and to acceptance testing in the maintenance stage. A main decision is whether to use top-down or bottom-up debugging, and it is suggested that top-down debugging is more efficient if combined with some of the other techniques. All the techniques shown are independent of any particular language or debug software.

KEY WORDS Debugging Test output Test data Performance standards

INTRODUCTION

Debugging (or program testing) is the process of making a program behave as intended. The difference between intended behaviour and actual behaviour is caused by 'bugs' (program errors) which are to be corrected during debugging.

Debugging is often considered a problem for three reasons:

1. The process is too costly (takes too much effort).
2. After debugging, the program still suffers from bugs.
3. When the program is later modified, bugs may turn up in completely unexpected places.

In general, there are two sources for these problems: poor program design and poor debugging techniques. For instance, the problem of too costly debugging may be due to the presence of many bugs (poor program design), or to a debugging technique where too few bugs are found for each test run or each man-day (poor debugging technique).

In this paper we will solely discuss debugging techniques, assuming that program design is adequate. In other words, we will consider this situation: A program or a set of intimately related programs are given. They contain an unknown number of bugs. Find and correct these bugs as fast as possible.

Debugging is carried out through *test runs*: execution of the program or parts of it with carefully selected input (so-called *test data*). Execution is normally done on a computer, but in some cases it can be advantageous to do a 'desk execution' as we will explain later.

Most of the techniques presented in this paper have been used for many years at Regnecentralen in Copenhagen and have to some extent spread from there.

TOP-DOWN DEBUGGING VERSUS BOTTOM-UP

In bottom-up debugging, each program module is tested separately in special test surroundings (module testing). Later, the modules are put together and tested as a whole (system testing).

In top-down debugging, the entire program is always tested as a whole in nearly the final form. The program tested differs from the final form in two ways:

0038–0644/79/0109–0051$01.00 *Received 3 July 1978*
© 1979 by John Wiley & Sons, Ltd.

1. At carefully selected places, output statements are inserted to print intermediate values (*test output*).
2. Program parts which are not yet developed are absent or replaced by dummy versions.

With this technique, test data has the same form as input for the final program.

Below I will argue that top-down debugging is more efficient than bottom-up, but first I will illustrate the principles by a case story.

Case story 1

A team of five programmers were implementing an 8-pass Algol compiler. Each pass was a rather self-contained module which input from disc the text string produced by the previous pass and output a transformed text string to be used by the next pass. The input to pass 1 was the source program and the output of pass 8 was the binary object program. The intermediate text strings were sequences of binary integers.

The plan was that debugging should be top-down, i.e. in principle the entire compiler should be used in each test run, and test data should be real source programs. Test output could be produced from each pass and consisted solely of a decimal copy of the output text string. With this set-up, pass 1 could be debugged with all the remaining passes absent, but pass 8 would require that all the preceding 7 passes worked reasonably well.

All passes were programmed at the same time, so the programmer responsible for pass 8 was worried about having his debugging severely delayed. Then he decided to use bottom-up debugging; he programmed an artificial surrounding and wrote artificial input for pass 8. This took a substantial time and when finally he had the surroundings working and had found the first bugs in pass 8, all the preceding passes were able to process the first simple source programs. So he could have spared the effort. Further, it turned out that pass 7 and pass 8 did not agree on the data formats communicated, a fact that in bottom-up debugging would have been concealed until very late.

The main difference between top-down and bottom-up debugging is that in top-down all test data has the form of *real input* to the final program. For bottom-up debugging this is only the case during system testing.

The advantages of top-down debugging can be summarized as follows:

A1. There is no need for artificial test surroundings. This saves much programming effort.

A2. Misunderstandings between communicating modules are revealed early in the debugging process. This spares debugging from wrong premises and avoids hostility between programmers in the system test stage.

A3. The test data used during debugging can easily be used during later maintenance to make sure that the program works at least as well as before (see the section 'Acceptance test').

The drawbacks of top-down debugging seem to be these:

D1. Sometimes it can be more difficult to test facilities of an 'inner' module. For instance, a complicated pattern of real input may be necessary to invoke the facility. But in principle it is always possible to test the facility through real input; otherwise the facility would be superfluous.

D2. Debugging of a particular module may be delayed because other modules must be debugged first. The sections 'Finding several bugs . . .' and 'Team debugging' give hints for reducing the delay. In Case story 1, pass 8 had to await partial debugging of the rest, but even in this case bottom-up debugging was no advantage.

D3. When a team of programmers work on the same program, one of them may easily

introduce an error that causes the program to fail for all test data. This will prevent the other programmers from debugging their parts for a while. A remedy for this *interference problem* is explained under 'Team debugging'.

The interference problem becomes worse with an increasing number of programmers, and there may be a point where bottom-up debugging becomes advantageous. However, in that case the 'modules' should be very large—about the work of 10 programmers—and each module should be debugged top-down.

In the literature there seem to be divided opinions on top-down or bottom-up. Several highly successful projects have been debugged top-down (Naur,[7] Baker,[1] and Lauesen[6]).

Brinch Hansen[2,3] is apparently using bottom-up debugging, but as a result no acceptance test exists. Brooks[4] reports on module test followed by system test, each stage occupying about one-quarter of the total development time. (This suggests that a large part of the debugging time, the system test, is performed in a top-down manner even though the basic approach is bottom-up.)

In tutorial papers some authors suggest top-down (Yohe[9]), others suggest bottom-up (Naur,[8] in spite of Naur's successful top-down experience[7]).

TEST OUTPUT VERSUS DUMP AND TRACE

In principle, the final output of the program is sufficient to reveal whether the program is correct (supposing, of course, that the test data is carefully selected). But the final output is insufficient to help you isolate the bugs. Additional output can be obtained through 'dumps' or through 'test output'.

A dump is produced after the program failed. It shows the contents of all or selected parts of the store. From this, the programmer is supposed to be able to find the bug. This method is inefficient for several reasons:

1. The bug will often have occurred long before its effects caused the program to crash, and thus the bug is well hidden in the dump. Finding more than one bug from a single dump is even more difficult.
2. Too much information is available in the dump and as a result the programmer wastes much time in studying the dump. If he is working in a high level language without 'formatted dumps', the relation between his source program and the dump is very obscure.

There are cases where a dump is the only practical way to find the bug, for instance if the bug caused program parts to be overwritten. Overwriting caused by hardware errors must also be located through dumps.

In normal debugging, a better technique is to get test output from the program while it is executing. Test output is produced by ordinary print statements inserted at carefully selected places of the program. The test output shows values of important variables, so that the progress of the execution can be followed afterwards.

Beginners tend to insert a lot of test output statements to make sure they get the essentials. However, if the program is properly structured, a few places will suffice even in large programs.

For instance, many programs have an *interpretative structure* like this:

```
          repeat
            get next input;
            find type of input;
            case type of
action1:      handling of type 1 input;
```

```
    action2:    handling of type 2 input;
                ...
              end;
            until last input;
```

The beginner will insert test output in each of the actions 'handling of type x input'. A better solution is to print all the essential variables in one statement just before '**case** type **of**'.

Another example appears in Case story 1: all passes produced their output on disk through a common subroutine which could print the 'character' in decimal form too. This single test output statement was sufficient for debugging the entire compiler (Naur[7]).

As a final example consider the following case story.

Case story 2

A team of seven programmers implemented an operating system. The operating system was structured as a set of parallel processes communicating with each other through a kind of 'message passing' using two subroutines: 'send message' and 'wait message'. Each process had an interpretative structure as above with 'get next input' implemented as 'wait message' and results produced in the actions through 'send message'.

Test output in this operating system was confined to two kinds:

1. 'Wait message' and 'send message' printed the contents of the message received or sent.
2. Each process printed essential variables once in the interpreting loop.

(For very complicated actions, it was necessary to print additional test output.) Each test output statement printed one line and the different kinds of test output were distinguished through a short text and a numerical identification of the process.

A test run using test output proceeds as follows. The program is corrected, compiled and then executed with test data. The result is a few pages of test output, probably mixed with real output from the program. At his desk, the programmer now studies the output noting whenever a wrong result appears. Then he knows that a bug must recently have been executed. Isolating this bug is normally very easy.

If the effect of the bug is not too serious, the programmer continues studying the output bearing in mind the bug he has found already. Normally, he will find other unexplained results and then he looks for more bugs. With this approach it is quite common to find a few bugs in one test run.

A criterion of good test output is that every bug causing a wrong result can be isolated without additional test runs to print more intermediate results. In my experience this can be obtained in most cases.

When the program is debugged, the test output statements are not removed. They can be conditional statements or they can write on a backing store file during production runs. For instance, all test output statements can look like this:

if test **then** print (...)

The variable 'test' is simply initialized to 'true' or 'false', or it is specified in the initial input.

In large programs it may be advantageous to select test output printing for different modules independently.

In Case story 1 it was possible to get test output optionally from individual passes. For instance, when debugging pass 7, only test output from passes 6 and 7 were printed, and this saved a lot of printer time.

The purpose of leaving test output in the final version is to facilitate maintenance. For real-time programs it can be essential to be able to find a bug in the very production run where it appeared. In that case, test output is stored on a backing store file for inspection if an error should occur.

In summary, test output gives the following advantages:

1. The programmer sees only meaningful results from the test run and in a format easily readable to him.
2. The bug will have occurred soon before the wrong result, which makes bug isolation rather easy.
3. Debug planning and bug isolation are done at the desk and not at the computer.
4. Test output can be left in the production version to facilitate maintenance.
5. The technique is independent of language and special utility packages.

To get the full benefits of the test output technique, it should be part of program design—not an afterthought.

Most computer manufacturers provide not only core dump facilities but also 'debug programs'. A typical debug program can print traces of the program, for instance a list of all jumps. It can also print selected parts of the store whenever execution passes certain 'break points'. These facilities can replace test output during debugging, but they cannot normally be left in the production version. As they rely on absolute machine addressing they also tend to move debugging from the desk to the computer. My conclusion is that they cannot compete with the modest effort required for proper test output.

TEST DATA SELECTION

The three problems of debugging mentioned in the Introduction give rise to three criteria for choosing test data:

1. Test data should reveal all possible bugs.
2. Test data should enable the programmer to find several bugs in each test session.
3. In the maintenance stage, it should be possible to confirm that the program still deals properly with old data (acceptance test).

Below we will discuss these requirements individually.

Revealing all bugs

It is well known that testing can only show the presence of bugs—not their absence. So, in general, it seems impossible to construct test data which can reveal all bugs.

Yet experience shows that thorough testing can reveal nearly all bugs. The reason seems to be that most bugs are simple mistakes like using a wrong constant, testing for 'less' instead of 'greater', etc. Such mistakes are revealed through internal testing (or 'path testing') where each branch of the program is tried.

The internal test will also reveal a large part of the non-immediate bugs which reflect misconception of the problem (for instance, a missing branch). More non-immediate bugs can be revealed by an external test where 'special' cases of the problem are tried to see whether they are properly dealt with in the program.

Goodenough and Gerhart[5] have outlined the possibility of a complete external test based on a problem definition independent of the program. They also point out that program proving cannot replace debugging, because the problem definition is hard to make complete and even harder to formalize into predicates. Manual program proving also fails to reveal many simple mistakes; for instance, a constant used in the proof could be mispunched in the program.

We will now explain three kinds of test: the internal and external test, as mentioned above, and the user test. Experience shows that few—if any—bugs remain if all three tests are carried out thoroughly.

1. Internal test. Here the programmer constructs test data so that all branches of the program are executed at least once. This is a rather mechanical task. Further, it is required that each branch gives a visible contribution to the output, and that is a little more difficult (see example below).
2. External test. Here the programmer constructs test data which tries all cases considered 'special' regarding the purpose of the program. Often all these cases are already part of the internal test, but sometimes the external requirements have special cases which are not reflected as special parts inside the program (see calendar example below). Preferably, the external test data should be constructed by someone who has not made the program.
3. User test. Here the user tests the system to see that it actually produces what he expects. The reason is that no matter how well the programmer has tested the program, he may still have misunderstood what the user expects. Only the user can check that.

These three kinds of test could be mixed into one set of test data. In this way all the cases common for internal and external test need be tried just once. But a combined internal and external test will not convince the user. He tends to consider these cases abnormal and of no concern to him. So normally the user's test data will be completely separate from the rest of the test data.

In the internal test we have to make sure that each branch of the program gives a visible result. This can be difficult because a given result may be caused by several branches. As an example, consider this interpretative loop:

```
          repeat
            get next input;
            find type of input;
            case type of
action1:      print ("a");  state: =1;
action2:      print ("a");  state: =2;
              ...
            end;
          until last input;
```

Here, the result 'a' is no indication that action 1 was executed. We also have to make sure that 'state' became 1. In principle, this can be revealed by additional input which gives an end result reflecting whether 'state' became 1. But often it will be simpler to use some test output to show the difference, for instance like this:

```
          repeat
            get next input;
            find type of input;
            if test then print ("place 1", type, state);
            case type of
action1:      print ("a");  state: =1;
action2:      print ("a");  state: =2;
              ...
            end;
          until last input;
```

This program will be easy to debug and will at the same time make it easy to check that a complete internal test has been carried out.

The statement 'all branches are executed at least once' may need a further explanation. In the interpretative loop it is obvious that all actions must have been executed. But in the statement

if c **then** S;

we should try two branches: c false and c true. If we try only c false, the statement S may be all rubbish. If we try only c true, the following statement may rely on something done in S, but may fail if S is bypassed.

Similarly, in the statement

if b **or** c **then** S;

we should try three 'branches': b and c false, b false and c true, b true and c false. In a loop like

for i: = 1 **to** n **do** S;

we should try two 'branches': $n < 1$ and $n > 1$.

In table-driven programs much of the logic is hidden in the tables. As a result, test data must ensure that all table entries are tried at least once. As an example consider a program which reads characters like this:

```
            repeat
              c: = next input character;
              type: = table[c];
              if test then print ("place2", c, type, . . .);
              case type of
digit:          handle digit;
letter:         handle letter;
                ...
              end;
```

Test data should not only ensure that the actions 'digit' and 'letter' are performed. It should also ensure that each element of 'table' is tried at least once. In other words, each character should appear at least once in the test data.

A very useful tool for internal testing is a program that supervises the execution of the program being debugged and reports on all branches not taken. In higher level languages such a tool could be a preprocessor which inserted suitable subroutine calls in each branch. Just before program termination a condensed report would be printed.

The so-called 'use profile' systems can also assist, as they show how often each statement has been executed. (In debugging we are looking for statements not executed at all.) But they cannot reveal missing branches, for instance whether

for i: = 1 **to** n **do** S;

has been tried with $n < 1$.

Note that such automatic tools are aids only. You still have to check manually that all output from the test run corresponds to the intended behaviour of the program.

In the external test we look at the purpose of the program and find 'special' cases and extreme cases. For instance, the program might not test for large input values and consequently such extreme cases would not be tried in the internal test. Still, overflow could cause wrong results. In external testing, you would try such extreme cases.

As another example, consider a special case which is not reflected in the program logic. A program might, for instance, convert month number 'm' and date 'd' to day number 'n' in the year according to this algorithm (leap years are ignored):

```
x: = m × 30.58 − 32.18 + d;
if m < 3 then n: = round (x + 1.6)
         else n: = round (x);
```

All the special cases of individual months are removed in the program and would not appear in an internal test. Yet, a proper external test should test first and last day in each month.

The principles of internal and external testing are described by Naur[8] and thoroughly discussed by Goodenough and Gerhart.[5]

Finding several bugs in each test session

In many cases a single set of test data could comprise all internal and external test cases. This would be convenient for an acceptance test in the maintenance stage. But in the initial debugging stage it would be awkward. The reason is that one test run normally reveals only one or a few bugs. If computer access is sparse, better performance is needed.

The proper procedure is to use several small sets of test data, each set trying only a small part of the program. In the interpretative loop, for instance, we could have one set of test data trying 'action1', another trying 'action 2', etc.

A test session would now proceed as follows. The program is corrected and compiled. It is then executed once with each set of test data. At his desk the programmer now studies the corresponding sets of test output, and he will most likely find one bug in action 1, another one in action 2, etc.

It turns out that with most types of programs it is possible to construct similar independent sets of test data, each one exploring only a part of the program.

In the very first test sessions, the programmer will probably only find bugs in the initialization part and central parts of the program. It is difficult to improve on this in the debug stage, but sometimes it can be taken into account during program design, for instance by distributing the initialization to independent program parts or organizing the initialization so that certain parts can be dummy in initial debugging.

For debugging of central program parts, 'desk execution' can be very useful. In desk execution, statements are simulated manually one by one. The value of variables are kept track of on paper with one column for each variable, so that the bottom value is always the current value of the variable. In a program with an interpreting structure, the loop—omitting all actions—is a candidate for desk execution. In machine language programming, the test output mechanisms themselves are candidates for desk execution.

If a team is debugging top-down, some effort should be made to get simple cases working first. In Case story 1, all 8 passes were very soon in a state where they could compile simple programs. In this way each pass could be debugged using the result of the previous passes. Detailed debugging of individual passes could proceed rather independently of other passes.

Acceptance test

Assume that a program is in the maintenance stage and a small change is made to it. How can we make sure that the modified program still deals properly with old data ? In principle it is simple: just run the old sets of test data and make sure that the results are the same—except where the modification should cause new results.

In 2011 terminology, this is called regression test. Acceptance test is the customer's accept of the system. It includes blackbox testing.

However, it is very cumbersome to check all results, and experience shows that the maintenance programmer tends to give it up or just makes a cursory inspection. Two improvements can be suggested:

1. Combine—as far as possible—the original sets of test data into one or a few acceptance tests.
2. Produce a short summary of each acceptance test.

A summary can be a sum and double sum of all test output and/or real output. Each item in the output could contribute to the summary like this:

```
sum: =(sum+item) mod max;
double sum: =(double sum+sum) mod max;
```

A deviation in an output value will show up in 'sum' and a deviation in sequence of values will show up in 'double sum'. Of course, two or more deviations may cancel each others contribution to the summary, but that is very unlikely. Acceptance testing by means of a summary was suggested by Naur.[8]

The summary can be produced either as part of the test output statements of the program during execution, or by a separate program which later scans the output and produces the summary.

If the program is a compiler or an operating system, the test data are other programs (test programs). Verifying the compiler or operating system means verifying that the test programs execute correctly. The test programs can be constructed so that they make a summary of their own execution. Compilation of an if-statement could, for instance, be summarized by this piece of the test program:

```
if i < 10 then p(1) else p(2)
```

Here, the procedure p prints its argument and includes it as an item in the sum and double sum.

When using a summary for acceptance testing, care must be taken to make the summary invariant to minor program changes. For instance, if the summary includes test output of machine addresses, a small program change may give a completely new summary. If test output is not included in the summary, but only 'real' output, care must be taken to ensure that the real output reflects all branches (see the section above, 'Revealing all bugs').

With acceptance tests, the maintenance programmer proceeds as follows before each release of a new version. He runs the acceptance test data on the new version and inspects the summary to see that it is just like the old version. He incorporates a test of the new facilities (or the bug corrected) into the acceptance test data and runs it for checking and to get a summary to be used as standard in the future.

It cannot be stressed enough that acceptance testing must be easy. The test data used in bottom-up testing is inadequate because it requires a splitting of the program into parts to be tested individually. Very few maintenance programmers will take that trouble or are given that time.

TEAM DEBUGGING

In top-down debugging or in the later stages of bottom-up debugging, testing a given module relies on other modules functioning reasonably well. When the modules are debugged by different programmers, one of them can easily introduce an error in his module that makes it completely non-functioning. As a result all the other programmers will waste their test runs.

A remedy for this interference is to work with two versions of each module: a project version which is always reasonably well-functioning (initially a dummy module), and a private version.

A programmer testing his own module would run the private version of the module combined with the project version of all other modules. If he finds the private version acceptable, he may decide to make it the new project version. However—and this is important—he must never take that decision in a hurry, least of all at the computer. He has to study the test results carefully at his desk before accepting the new version.

This approach has been successfully used in a team of seven programmers (Lauesen[6]). In a few cases it happened that two programmers introduced new versions that worked all right alone, but did not fit each other. This was rare, however, and could easily be fixed by returning to the old versions. (This was one reason for keeping back-up versions of all project files.)

MULTIPROGRAMMING

In ordinary sequential programs, errors are easy to reproduce. For instance, if a user complains about something not working, you can just ask for his input and process it with test output on. If the error cannot be reproduced, there are two possibilities: the environment (hardware, operating system, etc.) does not work, or there is an uninitialized variable in the program. (The latter possibility can be prevented by always clearing the entire store area of the job before loading the program.)

With multiprogramming (parallel processes) the situation is completely different: the process cannot be reproduced exactly. There are two reasons for this:

1. Certain actions are time dependent.
2. The total input material is not available for a rerun.

Time-dependent actions can be removed with proper programming techniques. Consider for instance Case story 2 where a set of parallel processes communicated by means of 'wait message' and 'send message'. These procedures corresponded to 'input' and 'output' in sequential programming, and as a result each process could be reproduced exactly if just the original sequence of messages was repeated.

Contrast this to a communication where 'wait message' could give two results: 'message received' or 'no message available'. Now we cannot reproduce the process by repeating the sequence of messages; we also have to issue the messages at the proper time. In other words, we have introduced time-dependent errors.

Another source of time-dependent errors is variables shared by two parallel processes. Here, the rule is to lock access to the shared variables while inspecting and updating them. If this rule is violated in the process, a wrong result will depend on timing relative to other processes.

But even if all time-dependent behaviour is eliminated, we cannot expect to reproduce a run, because we cannot get the total input material. For instance, the input to an operating system consists of all jobs processed and all commands typed in since the operating system was started. If one user complains about his job not working, we can get his data, but in practice we cannot get the remaining input.

The conclusion is that in the maintenance stage we have to catch errors while they occur. One method is to generate test output on a backing store during production runs. This approach was used in a large operating system (Lauesen[6]), and the result was a virtually error-free system about 6 months after the initial release.

Brinch Hansen[2, 3] also reports a virtually error-free operation of operating systems, but without test output during production runs. However, his operating systems were almost error-free in the first release, so the need for error detection in the maintenance stage was very small. (Reasons for the error-free release seem to be: (1) a very careful debugging phase,

(2) a much smaller operating system, (3) virtually no advanced strategies like deadline scheduling or dynamic resource allocation.)

In many real-time systems the backing store is so small that test output must be written cyclically in a file. If backing store is absent, a small area of the main store can be used for cyclical test output. This means that test output for the last period of execution will be available after a system crash. To avoid filling up the area with trivial test output, the highly repetitive program parts can execute without test output, but as soon as they detect an exceptional situation, they switch on test output.

The internal test in multiprogramming follows the same principles as in sequential programming. Each branch in each process must be executed at least once with a visible result. This means that we give up a systematic test for time-dependent errors, hoping that program design is adequate. Brinch Hansen[2] has shown an internal test of a multiprogramming kernel using just two test output statements. He also makes no attempt at finding time dependent errors.

A major problem when testing a multiprogramming system is to supply the input. The system often deals directly with peripheral devices, acting on exceptional status bits, etc. There are basically three ways to supply the input:

1. Brute force, e.g. typing the commands, loading mispunched cards, switching off power to a device.
2. Simulated devices, i.e. making two versions of all i/o subroutines: one which uses the physical device and one which reads/writes on a backing store instead of the physical device.
3. Hardware simulation, i.e. using a separate computer to generate the proper pulses on the device lines (Gould[10]).

Note that the brute force approach must be tried at least once to make sure that actual devices behave as assumed in the simulation methods. With the simulated device approach, the brute force method must be used whenever an i/o subroutine is changed.

PERFORMANCE STANDARDS

In planning the debug phase, you need some performance standards. The actual figures may vary from team to team and probably depend also on the kind of project. My own experience with many kinds of systems programming in several languages follows this performance standard:

1. One bug for each 20 lines of program.
2. One test session for each 40 lines of program.

A more accurate model for bug rates is provided by 'Software Science'.[11] A bug in this context means any error not detected by the compiler or assembler/linker. The majority of bugs are trivial ones like: a wrong constant, testing for $>$ instead of $<$, a forgotten statement, a wrong variable. These bugs can be corrected immediately when located.

Quite many 'bugs' reflect minor changes in specification. For instance, an additional blank in the output looks nicer, a check for duplicated input is convenient.

A few bugs reflect flaws in design of the algorithm. When you correct such a bug you feel you have to think a little. My own standard for such bugs is:

3. One non-immediate bug for each 150 lines of program.

To me it is a surprise to hear other programmers report much lower bug rates (e.g. Brinch Hansen[3]: about one bug for each 500 lines of program). For instance, I cannot see any way to avoid writing wrong constants now and then. One group reporting a similar low bug rate

explained that just before debugging they had two programmers checking the entire program, statement by statement. In their opinion the bug rate before this process might well have been one bug for each 20 lines, but the checking caught most of them. The man-hours used for checking seemed to be somewhat lower than the man-hours that would have been used for bug isolation after test runs.

No matter whether your bug rate is high or low, it is important to know it for planning purposes. A simple way to record bugs is to use the same program listing for a long period. When you have got the first compilation without errors, use the corresponding listing as a reference. Note all corrections on it. If you get a fresh listing for each run, do not use it for marking corrections. It may be convenient once or twice to nominate one of these listings as a new reference, but keep all the reference listings. At the end of the project it is now easy to count and classify the bugs.

The number of test runs in my case corresponds to finding an average of two bugs for each test session. Note that a test session comprises execution of several sets of test data each exploring independent parts of the program. Actually, the number of bugs per session varies as follows. The first sessions on a 'fresh' module reveal less than one bug each (syntax errors and bugs in initialization and central parts). The next sessions reveal around five bugs each (taking into account that a syntax error in a correction now and then spoils a session). In the last sessions, bugs are sparse and only one bug is revealed per session.

Surprisingly, the much lower bug rate reported by Brinch Hansen does not result in fewer test sessions. He used 31 sessions to debug 22 pages of program.

If you know your performance standards, you can make a rather precise plan for the debug phase. For instance, if I have a program of 2,000 lines, I know there will be about 100 bugs and 50 test sessions will be needed. So if I can expect two test sessions a day, about 25 working days are required for debugging. With one test session a day, 50 days are needed.

With four test sessions a day, 12 days seem sufficient, but alas, I know I cannot isolate 4×5 bugs a day, as required in the middle part of debugging. Most likely the result would be a need for more than 50 test sessions.

DIFFICULT THINGS TO TEST

In the section on 'Test data selection' we discussed the techniques of internal and external testing. In my experience there are types of bugs that are difficult to find with these techniques.

As the first example, consider a table-driven program where many table entries mean 'illegal input', but have the same effect: an error message. Typically, a compiler driven by a state table could have several hundred such error entries. In internal testing we require that each table entry is tried at least once, but in practice it can be prohibitive to try all error entries. The result of such an incomplete test could be a program which responds properly to all legal inputs but not to all illegal inputs.

As the next example, consider a program part which tests a certain bit and chooses a path accordingly. The internal test has shown that the paths are followed properly in the proper cases. However, the program actually tests a completely wrong bit, which just happened to match the value of the proper bit in the cases tested. (Such a match is quite likely if each path is tried just once.) The bug cannot reliably be found through internal testing, and there is nothing in the program specification to suggest an external test case which can reveal the error.

This bug was an example of 'wild addressing'. Another case of wild addressing is seen when a program stores an intermediate result using a wrong address. The final results may be correct as long as the spoiled parts of the store are not used later in the process. Again, there is

no systematic way of detecting the bug. It is some help to use a high-level language with an index check, because that prevents spoiling of program parts; but wrong variables can still be addressed explicitly.

The last example is only seen in multiprogramming: updating a shared variable without proper locking. Language constructs associating shared variables with 'critical regions' definitely help here, as locking is done automatically. However, it is too late to apply this technique in the debug stage, so it is outside the scope of this paper.

The existence of 'difficult' bugs stresses the need for being able to reproduce a production run with test output (monoprogramming) or catching the test output during the production run (multiprogramming).

CONCLUSION

We have described several techniques for improving debugging:

1. Top-down debugging.
2. Test output inserted a few central places.
3. Conditional test output statements which can be left in the production version.
4. Test data comprising internal test cases, external test cases, and user test.
5. Many small sets of independent test data.
6. A combination of test data into an acceptance test with summary.
7. Keeping a project version and a private version of each module in team debugging.
8. Collecting test output during production runs for multiprogrammed systems.
9. Device simulation for multiprogrammed systems.
10. Keeping records of bugs for comparison with performance standards.

REFERENCES

1. F. T. Baker, 'Chief programmer team management of production programming', *IBM Syst. J.* **11, 1**, 56–73 (1972).
2. P. Brinch Hansen, 'Testing a multiprogramming system', *Software—Practice and Experience*, **3,** 145–150 (1973).
3. P. Brinch Hansen, 'The Solo operating system: processes, monitors and classes', *Software—Practice and Experience*, **6,** 165–200 (1976).
4. F. P. Brooks, *The Mythical Man-month*, Addison–Wesley, London, 1975.
5. J. B. Goodenough and S. L. Gerhart, 'Toward a theory of test data selection', *SIGPLAN Notices*, **10, 6**, 493–510 (1975).
6. S. Lauesen, 'A large semaphore based operating system', *Comm. ACM.* **18, 7**, 377–389 (1975).
7. P. Naur, 'The design of the Gier Algol compiler, Part 2', *BIT*, **3,** 145–166 (1963).
8. P. Naur, *Concise Survey of Computer Methods*, Studentlitteratur, Lund, 1974.
9. J. M. Yohe, 'An overview of programming practices', *Computing Surveys*, **6, 4**, 221–243 (1974).
10. J. S. Gould, 'On automatic testing of on-line real-time systems', *Proc. AFIPS, Spring Joint Computer Conf.*, 477–484 (1971).
11. A. Fitzsimmons and T. Love, 'A review and evaluation of software science', *Computing Surveys*, **10, 1**, 3–18 (1978).