Testing and diagnostic aids for real-time programming

E.C. SEDMAN
Computer Analysts and Programmers Ltd, Reading, England

1. Introduction

Most papers on computing topics tend to be concerned with the design of programs or systems, but experience indicates that this is only a small part of the total effort involved. Aron [1] states that 30% of the elapsed time of a large project is spent on design, with 40% on implementation and 30% on testing. Figure 1 is a simplified form of Aron's diagram, which shows how the number of people engaged in a project changes during the life of the project. In terms of resources, the ratios are probably more like 20%, 40% and 40%, so it is worthwhile to consider the aids available to improve performance in these latter stages.

Real-time systems introduce a number of special problems, and this paper discusses them under two main headings. Section 2 deals with testing, i.e. getting rid of bugs before the system goes live. Section 3 deals with diagnostic aids, i.e. finding the bugs which were not got rid of before the system went live.

2.1 Test environments

It is common practice to create a test system to facilitate the unit testing of individual components and also the testing of functional packages. Such a tool has been described by McKenzie and Weston [2]. While not peculiar to real-time programming, it is particularly valuable for this because bugs which are not found at the unit testing stage can be very difficult to find at a later stage.

A particular point is that the test system should not zero out blocks of core before they are given to the program under test, if this will not happen in the live environment. It is even better to overwrite the block with some arbitrary pattern when it is returned to the system. This will help to detect errors resulting from references to data that is no longer available. One should generally try to prevent future trouble that could arise from implicit assumptions about control program interfaces.

The use of programming standards and conventions, for example in the use of registers, is similarly not special to real-time. Their use is essential however because of the large number of interactions that can take place.

2.2 Special hardware

In process control and similar applications, one is frequently dependent on special hardware such as analogue to digital converters, non-standard interfaces, etc. A good general rule is not to trust it - the combination of untried software and untried hardware can be a nightmare.

When starting system testing under such circumstances it is often useful to drive the programs using simulated inputs. The routine which services the special device is replaced by a dummy which supplies predetermined values. A good example of this used the standard service routine to issue a 'read' and service the interrupt, but it then ignored the data so obtained and took values read in from the disk. In this way valid timings were obtained, but one could also reproduce a test run without having to worry about whether the hardware was behaving consistently. Of particular value was the ability to introduce incorrect values in a controlled way, in order to check that the programs behaved correctly in error situations.
On the output side the same thing applies: the fact that equipment is not behaving in the way that it should, does not necessarily mean that the wrong values are being output by the program. It is therefore worthwhile to have a means of recording the values being output to a device.

One of the most sophisticated simulations of this type has been used in the Concorde fatigue test, where independent computers are used for control and monitoring. To test the system out, the computers were run 'back-to-back', with the output from the control computer fed directly to the monitoring computer.

As well as providing simulators to drive the software, it is also useful to have simulators to drive the hardware. This mean essentially simple programs to exercise the hardware and record the results. Such programs often have an extended life for diagnostic purposes long after the initial testing stage, but they are often best devised by a hardware expert who understands the way the hardware works. A further point to be borne in mind is that the diagnostic programs should be able to be time-shared with other work, or one can find that the whole computer is being used to check out one minor device.

2.3 Terminal driven systems

In the case of systems with a large number of on-line terminals, one has a different hardware problem. Either the hardware is not available when the system is being developed, or, if it is, one does not have the staff to drive all the terminals at once. Even if one had, there would again be the problem of lack of reproducibility, so again one has to provide a means of simulating inputs.

On one such system, it is possible to run card based tests of the version under development, concurrently with terminal usage of the live system. For this particular case, only single-thread testing is currently available, but multi-thread testing is desirable. The problem here is that concurrent tasks will not always terminate in the same order on successive runs, due, for example, to differences in disk arm movement. It is especially important to check that the correct action is taken when a task fails to obtain a resource. This is a special case of the general testing of limit conditions and should include looking for 'deadly embrace' situations. Volume testing of any exclusive read facility is desirable, while dealing with the maximum number of simultaneous transactions will help to show up any non-reentrant coding.

2.4 Error recovery

An important area that is often insufficiently tested is that of error recovery and restarting. The problems of providing a 'warm restart' can easily be underestimated, and an error arising during this procedure can be a major difficulty. Reconfiguration after an error must be checked out, and a requirement to run in degraded mode implies that such running must be tested.

It is often necessary to provide a file recovery program. This enables records that have been lost or corrupted to be repaired. When such a program operates on records that are chained together, it is particularly important to test that no harm comes from an attempt by another program to access a record which is currently undergoing repair.

Finally, it is necessary to test the recovery from hardware errors as well as software errors. For example, the action taken in the event of a disk read failure is often not tested until such a failure occurs. To produce a reliable system, it is necessary to check out all such actions (perhaps by simulation) before the error arises in practice.

3.1 Tracing

One of the simplest, but often the most useful debugging aid is a list in order of, say, the last 20 routines called before the crash. If one has a standard method of entering routines, there is very little overhead involved in storing in a circular buffer a code number for each routine as it is entered. It is usually worthwhile to keep such a list for each task.

A more elaborate facility is a dynamic trace that can be activated or deactivated selectively from a terminal. Its output can be written to magnetic tape for subsequent off-line analysis. Again the facility has to be built into standard entry and exit procedures. It involves some overhead, but its great advantage arises if it can be initiated selectively when there is trouble, without the need for any recompilation.

Sometimes the only way to find a bug is to introduce snapshots in the routine where the trouble is thought to be. This involves recompilation of course, but can be made easier if a standard routine exists which can be simply called.

3.2 Consistency checks

With a multi terminal system, one can differentiate sharply between errors which affect one user, and errors which affect the whole system. For errors in the first category it is usually possible to 'kill' the user concerned, but to keep the rest of the system working. Errors in the second category usually bring the whole system down. It is therefore good practice to carry out consistency checks on all data handed to the control programs, and if possible to carry out the checks at as early a
stage as possible in the application programs. Failure of such a check results in a selective dump of information about the particular user and the termination of his task.

A typical check can make use of a record identifier. When reading in a record from disk an identifying character is checked to ensure that the record has not been corrupted. Equally, before writing out a record, the character is checked as a protection against the record having been overwritten in core. Similarly, it is possible to have a block identifier. When a program is given a core block by the control program, it can check that its address is in an acceptable range, and also check a character in the block which indicates its size. In this way one can avoid problems arising from blocks being put onto the wrong lists when they are released.

Again, when updating a record the program doing so must have exclusive access to the record, i.e. it must 'hold' the record. It is possible to keep a check on exclusive reads, and ensure that this is the case before updating.

In designing a real-time system, redundant information of this type should be inserted deliberately to provide additional checking.

3.3 Monitoring

One of the problems with the consistency checks mentioned in the previous section is that the damage may have been done some considerable time before it was detected, with the result that it may be very difficult to find the cause of the trouble. To overcome this, a monitor can be introduced, which carries out checks at regular intervals, to try to detect corruption as soon as it occurs. The best chance of pinpointing the fault will be if the check is carried out on every subroutine entry, but it is adequate to check each time a task is scheduled because another has terminated or gone into a wait. Under these circumstances the task causing the trouble can be determined uniquely, and hence from the trace one can obtain the sequence of routine calls responsible.

The information to be checked can be anything which is known to become corrupted. Typically this might be a chain of core blocks each of which contains pointers to the next one and the last one. It is possible to chase down such chains, carrying out a consistency check, and when an address is incorrect it is known that overwriting has occurred. This is another example where use is made of redundant information designed into a system. In one case, the use of this technique rapidly identified a major bug which had been causing trouble but had remained unsolved for several months. It also detected four other bugs whose presence had been suspected but not identified for an even longer period.

3.4 Footprints

When a common data base can be updated by many different application programs there are always garbage disposal problems, but more serious are the problems of records which have been disposed of when they are still required. To check on this it is possible, each time a record is released, to log information about the program doing the releasing. In this way one can obtain evidence about programs which release records that are still required.

4 Conclusion

The above has been in the nature of a review of testing and diagnostic aids. It is hoped that it will be of some use as a check list, and also perhaps encourage the dissemination of other ideas that could be generally used.


Discussion

Q. Which computers did you use for the Concorde fatigue test, and which languages did you use to program them?

A. There are three computers. Control of the loads is by a PDP-8. Control of pressures, temperatures, etc. is by another PDP-8 with special hardware and software supplied by Kent Instruments. The monitoring computer is a 48K word DEC system 10 (PDP-10).

The most difficult programming is on the monitoring computer. We did detailed design of the programs in PL/1, and they were translated by hand to Macro-10 by a more junior programmer, who also tested them. This worked fairly well because two people worked on every program, and there was some element of competition because the junior tried hard to find logical errors in the design given to him. It was not completely successful because corrections tended to be made to the Macro-10, without the PL/1 being updated.