Reproducible Testing of Monitors*

(1978)

This paper describes a systematic method for testing monitor modules which control process interactions in concurrent programs. A monitor is tested by executing a concurrent program in which the processes are synchronized by a clock to make the sequence of interactions reproducible. The method separates the construction and implementation of test cases and makes the analysis of a concurrent experiment similar to the analysis of a sequential program. The implementation of a test program is almost mechanical. The method, which is illustrated by an example, has been used successfully to test a multicomputer network program written in Concurrent Pascal.

1 Introduction

Some computer scientists feel that testing is a futile effort since it can never convince one that a program is absolutely correct. My experience has been that systematic testing can be remarkably successful in practice for both sequential and concurrent programs (Naur 1963; Brinch Hansen 1973, 1977). I am sure that there are good theoretical reasons for this which could be uncovered if researchers would take a fresh look at the problem.

The purpose of this paper is to describe a method for testing monitor modules which control the interactions of processes in concurrent programs. The testing of a monitor involves several steps:

1. For each monitor operation the programmer identifies a set of preconditions that will cause each branch of the operation to be executed at least once.

2. The programmer then constructs a sequence of monitor calls that will try each operation under each of its preconditions.

*P. Brinch Hansen, Reproducible testing of monitors, *Software—Practice and Experience 8*, 6 (November–December), 721–729.

3. The programmer now constructs a set of test processes that will interact exactly as defined above. These processes are scheduled by means of a clock monitor used for testing only.

4. Finally, the test program is executed and its output is compared with the predicted output.

This method makes it possible to test a monitor module by means of a reproducible sequence of operations without changing the monitor. The method has been used successfully to test a network program for a multicomputer system (Brinch Hansen 1978). The test example described in this paper was carried out successfully on a PDP 11/55 computer.

2 An Example: The Asymmetric Buffer

Figure 1 shows an example of two processes that communicate by means of a buffer monitor.

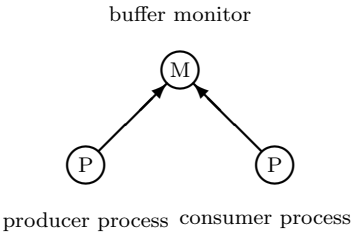

Figure 1 A concurrent program.

The arrows indicate that these processes have access to that monitor. A producer process inputs one *line* at a time from a card reader and sends it through a buffer. A consumer process receives one *character* at a time from the buffer. The buffer has a capacity of one line. When the buffer is full, 80 receive operations must be performed before another send operation can take place. The characters in each line must be received in their natural order (from left to right).

In the programming language *Concurrent Pascal* (Brinch Hansen 1977) this buffer can be programmed as follows:

```
type buffer=
monitor
var contents: line; length: integer;
  sender, receiver: queue;

procedure entry send(x: line);
begin
  if length > 0 then delay(sender);
  contents := x; length := 80;
  if not empty(receiver) then
    continue(receiver)
end;

procedure entry receive(var y: char);
begin
  if length = 0 then delay(receiver);
  y := contents[length]; length := length − 1;
  if not empty(sender) & (length = 0) then
    continue(sender)
end;

begin length := 0 end
```

The monitor defines a data structure representing the buffer and two operations, *send* and *receive*, on the buffer. An initial statement at the end of the monitor makes it empty to begin with.

The data structure represents the buffer *contents* (a single line) and its current *length* in characters ($0 \leq \text{length} \leq 80$). Two queue variables are used to delay the *sender* and the *receiver* processes (if necessary).

The send operation delays the producer process until the buffer is empty. It then puts a new line into the buffer. If the consumer process is waiting in the receiver queue its execution is continued.

The receive operation delays the consumer process until the buffer is nonempty. It then gets the next character from the buffer. If the buffer becomes empty and the producer process is waiting in the sender queue its execution is continued.

In Concurrent Pascal a *continue* operation has no effect if a queue is *empty*. It is therefore unnecessary to examine whether a queue is empty before performing a continue operation on it. The continue operations are nevertheless made conditional in this monitor to make it clear that they should be tested in two cases: when the queues are empty and also when they are not empty.

Processes can perform the send and receive operation on the buffer but cannot access the data structure representing it directly. This is guaranteed by the compiler.

The operations on a monitor take place strictly one at a time. When a process performs a monitor operation the computer will automatically delay further operations on the same monitor until the current operation is either finished or delayed explicitly in a queue variable. In the latter case, the delayed operation is resumed when another monitor operation performs a continue operation on the same queue.

(When I first wrote this paper the buffer monitor was programmed exactly as shown here. The test described in the following revealed a programming error. I found this very appropriate in a paper on systematic testing and decided to postpone the correction of the error until the end of the paper.)

3 Planning a Test Sequence

When a monitor has been programmed it must be tested systematically. The first step is to identify the *test cases* that must be tried experimentally.

In the buffer example both procedures may or may not delay the calling process and may or may not continue the other process. These are the only choices. So the following are the necessary *preconditions* that will ensure that all statements within the monitor procedures are executed at least once:

test cases (send)	*preconditions*
sender is delayed	S_1: length > 0
sender is not delayed	S_2: length $= 0$
receiver is continued	S_3: **not** empty(receiver)
receiver not continued	S_4: empty(receiver)

test cases (receive)	*preconditions*
receiver is delayed	R_1: length $= 0$
receiver is not delayed	R_2: length > 0
sender is continued	R_3: **not** empty(sender) & length $= 1$
sender is not continued	R_4: empty(sender) **or**
	R_5: length > 1

These preconditions must hold upon entry to the send and receive procedures.

It may be helpful to outline briefly how these test cases were identified. The aim of the testing is to try each branch of the monitor procedures at

least once. Take, for example, the receive procedure. Its first *if* statement must be tried both when the condition (length = 0) is true and when it is false. This observation immediately defines the preconditions R_1 and R_2.

The next two statements are assignments. Since they are executed unconditionally, it is not necessary to plan special test cases for these statements. They will always be executed.

The final *if* statement should be tried both when the compound condition is true and when each of its terms is false. This defines test cases R_3–R_5.

The test cases for the send procedure were found by similar reasoning.

One may ask if it is sufficient to test each branch once instead of testing all combinations of all branches. The answer is that exhaustive testing of all the possible paths through a program normally is impractical. As usual, one must depend on insight into the program structure to reduce the number of test cases drastically.

Now, after the first *if* statement in the receive procedure it turns out to be irrelevant whether or not the calling process was delayed. In both cases the buffer will eventually end up in the same state (length > 0). So, at this point, the state of the monitor variables is independent of the actual path of program execution. This "memoryless" property is characteristic of well-structured programs. If it is carried far enough it is indeed sufficient to consider the two conditional statements as separate (unrelated) test cases.

The *path independence* of structured programs applies not only to statements (including loops), but also to procedures, monitors, and entire programs. It is not necessary to test all possible sequences of monitor operations. All that is needed is one *representative sequence* of operations that covers the relevant state transitions.

The study of the programming principles that make systematic testing possible is a fascinating research area that has barely been touched yet. It is not the purpose of this paper to discuss these issues. We will merely point out that whenever it is difficult to identify relevant test cases in a system program one can safely assume that this is due to poor structuring. The planning of test cases can therefore serve as an indication of unnecessary complexity.†

In the following we will take it for granted that one can identify a small number of relevant test cases for a monitor. Our main concern then is to plan an experiment that will cover all the test cases once they have been

†In designing and testing a total of 5000 lines of Concurrent Pascal programs I have not yet found it necessary to write a monitor that exceeds one page of text.

identified.

We will therefore try to construct a set of processes that will test the buffer monitor systematically. To begin with we will, however, ignore these processes and focus our attention on what should happen within the monitor when it is being tested.

Since monitor operations are carried out strictly one at a time one can plan a sequence of operations that will force the monitor to go through all the test cases. One can then take the operations of this *test sequence* and distribute them among a set of test processes and make these processes perform the monitor operations in exactly the same order. Such a test will be both systematic and reproducible.

In practice, it is fairly easy to construct a test sequence by trial and error. The following shows one of the possible test sequences for the asymmetric buffer. The line length has been reduced to two characters for testing purposes. The producer process sends two lines consisting of the strings 'ab' and 'cd'. The state of the monitor variables before and after each operation are defined by *test assertions*. The assertions are labeled S_1, S_2 and so forth, to indicate which test cases they imply. The test sequence covers all the nine test cases described earlier.

{R_1, R_4: contents = [ ], empty(receiver), empty(sender)}
    receive( )
{S_2, S_3: contents = [ ], **not** empty(receiver), empty(sender)}
    send('ab')
{contents = 'ab', empty(receiver), empty(sender)}
    receive( ) continued
{S_1, S_4: contents = 'b', empty(receiver), empty(sender)}
    send('cd')
{R_2, R_3: contents = 'b', empty(receiver), **not** empty(sender)}
    receive( )
{contents = [ ], empty(receiver), empty(sender)}
    send('cd') continued
{R_2, R_5: contents = 'cd', empty(receiver), empty(sender)}
    receive( )
{R_4: contents = 'd', empty(receiver), empty(sender)}
    receive( )
{contents = [ ], empty(receiver), empty(sender)}

4 Implementing a Test Sequence

Although the approach taken here is informal one could use verification techniques to show that the test sequence has the effect defined by the assertions. In the present example, the result of the test seems fairly obvious. But, even if formal techniques are used, typing mistakes can still be made when the program text is entered into the computer. So one must still execute the test sequence.

In a concurrent program the relative progress of processes will normally be influenced by numerous unpredictable and irreproducible events, such as the exact timing of interrupts, the presence of other (perhaps unrelated) processes, the occurrence of transient input/output errors, and the speed with which operators interact with the program. The exact timing of operations will therefore, in general, vary somewhat from one execution of the program to another of a concurrent program even though the input remains the same!

During testing, however, we must be able to control precisely the sequence in which two or more processes interact with a monitor. Otherwise, we cannot be sure that all the test cases have been tried.

We will therefore assume that the processes synchronize themselves by means of an abstract clock during testing. This *test clock* is incremented by one after each operation (or partial operation) in the test sequence. The partial operations are the ones that are split into pieces by means of delay operations.

With this idea in mind we now rewrite the test sequence and attach a unique time value t_1, t_2 and so on to each operation:

t_1: receive( )
t_2: send('ab')
t_3: receive( ) continued
t_4: send('cd')
t_5: receive( )
t_6: send('cd') continued
t_7: receive( )
t_8: receive( )

The test clock is a standard monitor that implements an operation

clock.await(t)

which delays a test process until the time is t.

The *test processes* that implement the previous sequence of operations can now be programmed as follows:

```
producer:
  process(buf: buffer; clock: testclock);
  begin
    with buf, clock do
    begin
      await(2); send('ab');
      await(4); send('cd');
      await(6)
    end
  end

consumer:
  process(buf: buffer; clock: testclock;
    terminal: display);
  var c: char;
  begin
    with buf, clock, terminal do
    begin
      await(1); receive(c);
      await(3); print(c);
      await(5); receive(c); print(c);
      await(7); receive(c); print(c);
      await(8); receive(c); print(c)
    end
  end
```

The consumer process uses another standard monitor to print the characters it receives on a *display*. Notice that the construction of the test processes on the basis of the previous time sequence is an almost clerical task.

5 The Test Clock

The test clock is a monitor that implements two operations. An *await* operation delays the calling process until the clock reaches a certain value. A *tick* operation increments the clock by one and wakes up the process (if any) that is waiting for the new time value.

The current time is represented by an integer. Each instant of time is the starting time of a single monitor operation performed by a single process. Consequently, each time value in the test sequence can be represented by a queue variable that permits one (and only one) process to wait for that moment. The whole test sequence can therefore be represented by an array of queues.

```
type testclock =
monitor
var time: integer;
  sequence: array [1..steplimit] of queue;

procedure entry await(when: integer);
begin delay(sequence[when]) end;

procedure entry tick;
begin
  time := time + 1;
  continue(sequence[time]);
end;

begin time := 0 end;
```

How does one control the passage of time? The simplest idea is to let the test processes make the clock tick after each monitor operation. But this would not work since a process can be delayed by a test operation before it gets a chance to call *tick*. Since our purpose is to test the buffer monitor *as it is* we rule out the possibility of putting additional clock statements within it. (This idea would not have worked either since the language does not make it possible to combine a *delay* and a *tick* into a single indivisible operation.)

The progression of time must therefore be controlled by a separate process that makes the test clock tick at regular intervals. A clock interval of 3 seconds is used. This is an order of magnitude longer than the slowest operation in the test program (in this case, the printing of a single character on the display). Consequently, one test operation will always be completed before the next one begins.

```
timer:
  process(clock: testclock);
  var step, sec: integer;
  begin
    for step := 1 to steplimit do
    begin
      for sec := 1 to 3 do
        wait "one second";
      clock.tick
    end
  end
```

Since a test process only waits for times that are in the future and since the clock continues to increase (thanks to the timer), the test clock will eventually terminate the waiting of any test process and enable it to perform its next operation. The whole test sequence will therefore be carried out as planned.

6 A Complete Test Program

The buffer monitor is tested by means of the concurrent program shown in Fig. 2. The program consists of three processes (the producer, the consumer and the timer) and three monitors (the buffer, the test clock and the display controller). The arrows show how the modules call one another.

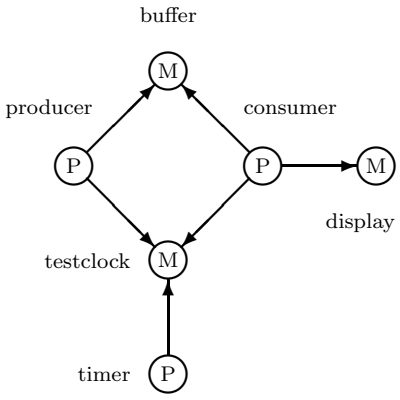

Figure 2 Test program.

The modules are initialized and linked to one another by an *initial statement* in the program:

```
var buf: buffer; clock: testclock; terminal: display;
begin
  init buf, clock, terminal, timer(clock),
    producer(buf, clock),
    consumer(buf, clock, terminal)
end
```

When the test program was executed on the PDP 11/55 computer it

printed the output 'badc' instead of the expected 'abcd'. An examination of the receive operation on the buffer made it clear that the statement

$$\text{y} := \text{contents}[\text{length}]$$

should have been

$$\text{y} := \text{contents}[\text{linelength} - \text{length} + 1]$$

After this correction the test gave the correct output.

7 Final Remarks

Since the operations on a monitor can only take place one at a time, one can construct a sequence of monitor operations that will make the monitor variables go through all the states which the programmer wishes to test.

One can then assign consecutive time values to these operations and construct a set of processes that will perform the operations in the specified order. These processes are synchronized by a clock that is very slow compared to the monitor operations being tested.

This method separates the construction and implementation of test cases and makes the analysis of a concurrent experiment similar to the analysis of a sequential program. The implementation of a test program is almost mechanical in nature.

If an experiment tests all possible process interactions that can occur within a monitor then that monitor will continue to behave correctly when it is used in a concurrent program in which the precise timing of events is not controlled by a central clock.

The task of constructing a separate test program for each module in a concurrent program is much simpler than one might imagine. The resulting reliability of programs makes it an essential and worthwhile effort. A detailed example of the test programs for a *real-time scheduler* is included in Brinch Hansen (1977). The *network program* described in Brinch Hansen (1978) also worked immediately after being tested by this method.

Systematic testing deserves to be studied more carefully by computer scientists. Such an effort could lead to a theory that would point out precisely under which circumstances program testing can be successful. It seems plausible that the program structures which simplify formal verification will also

simplify testing. Verification and testing would then emerge as complementary methods for obtaining program correctness. This important point was also made in Goodenough (1977).

Acknowledgements

I wish to thank Roger Vossler of TRW Systems, Redondo Beach, California, for providing the opportunity to develop the testing method described here. The Network program was originally tested at TRW's Signal Processing Facility. I also thank Nissim Francez, Charles Hayden and Jørgen Staunstrup for their constructive criticism of this paper.

References

Brinch Hansen, P. 1973. Testing a multiprogramming system. *Software—Practice and Experience 3*, 2 (April–June), 145–150. *Article 5.*

Brinch Hansen, P. 1977. *The Architecture of Concurrent Programs*, Prentice Hall, Englewood Cliffs, NJ, (July).

Brinch Hansen, P. 1978. Network: a multiprocessor program. *IEEE Transactions on Software Engineering 4*, 3 (May), 194–199. *Article 13.*

Goodenough, J.B., and Gerhart, S.L. 1977. Toward a theory of testing: data selection criteria. In *Current Trends in Programming Methodology II*, R.T. Yeh, Ed., Prentice Hall, Englewood Cliffs, NJ.

Naur, P. 1963. The design of the Gier Algol compiler. *BIT 3*, 2–3, 124–143 and 145–166.