1 Acceptance Tests

Acceptance tests are, simply put, a check of reasonableness, and fall into the following categories.

1.1  Timing Checks

If we have a rough idea of how long the code should run, a watchdog timer can be set appropriately. If the timer goes off before being reset, we can assume that a failure has occurred (say, the program has “hung”). The timing check can be used in parallel with other acceptance tests.

1.2  Results Checking and Correction

The material discussed in this section is derived from: M. Blum and H. Wasserman, “Reflections on the Pentium Division Bug,” IEEE Transactions, vol. 45, no. 4, pp. 385-393, April 1996. The paper is available via the course web site.

Simple checking is based on the observation that for certain mathematical functions $f$, the task of determining, given inputs $x$ and $y$, whether or not $f(x) = y$ is easier than the task of, on input $x$, computing $f(x)$. So, a checker for function $f$ is a program which, given inputs $x$ and $y$, returns the correct answer to the question, “Does $f(x) = y$?” The checker may be randomized, in which case, for any $x$ and $y$, it must give the correct answer with high probability over its internal randomization. Furthermore, the checker must take asymptotically less time than any possible program for computing $f$.

As an example, we will consider a simple checker for matrix multiplication. Let us consider two matrices $A$, and $B$. The product matrix $C = A \times B$ is computed as follows

$$A \times B = \begin{pmatrix} 1 & -2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & -1 & 0 \\ 2 & 2 & 2 \\ 3 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & -5 & -3 \\ 15 & 6 & 9 \\ 0 & -1 & 0 \end{pmatrix}$$

The naive way of multiplying two $n \times n$ matrices takes time $O(n^3)$. More sophisticated methods have been developed that take time $O(n^c)$ for various values of $c$, $2 < c < 3$. A program implementing such an algorithm could be quite complex and so, perhaps, buggy and in need of a checker.

The matrix-multiplication checker takes as inputs $n \times n$ matrices $A$, $B$, and $C$. It must determine whether or not $A \times B = C$. It begins by generating an $n$-high column vector $r$ of random numbers. It then calculates and compares products $A(Br)$ and $Cr$. Note that these calculations can be done in time $O(n^2)$, using straightforward vector-by-matrix multiplication. For example, we can pick a random value for $r$ and use it to check the matrix multiplication from above.
\[ A \times (Br) = \begin{pmatrix} 1 & -2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & -1 & 0 \\ 2 & 2 & 2 \\ 3 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix} \]

\[ = \begin{pmatrix} 1 & -2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} -2 \\ 0 \\ 0 \end{pmatrix} \]

\[ = \begin{pmatrix} -2 \\ 0 \\ -2 \end{pmatrix} \]

and

\[ Cr = \begin{pmatrix} -1 & -5 & -3 \\ 15 & 6 & 9 \\ 0 & -1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \\ -2 \end{pmatrix} \]

Clearly, if \( A \times B = C \), then \( A(\text{Br}) = (AB)r = Cr \) for any \( r \). However, if \( A \times B \neq C \), it is possible that, for certain values of \( r \), \( A(\text{Br}) \) may nevertheless equal \( Cr \). However, it can be shown that the probability (over the random choice of \( r \)) of the checker being fooled in this way is small. If we repeat the above process for several values of \( r \) and accept \( C \) as correct if \( A(\text{Br}) = Cr \) for all \( r \) tried, we have a simple checker whose probability of error can be made arbitrarily low.

We have here checked a complicated, \( O(n^c) \)-time program with a simple, \( O(n^2) \)-time program. The checker is thus easy to code compared to the program it checks, unlikely to be buggy compared to the program it checks, and quick to execute compared to the program it checks.

Now, we will consider the possibility that having detected its output to be incorrect, the program attempts to correct itself. Correction is based on the fact that, for many functions \( f \), we can efficiently compute \( f(x) \) if we know the value of \( f \) at several random inputs other than \( x \). So, it is possible to work our way around occasional “bad inputs” at which a program for computing \( f \) fails. Since it is easy to determine, via a simple stage of random testing, that such a program is correct on most inputs, a self-corrector of this sort will then suffice to patch over the remaining, occasional errors, making the program effectively perfect.

Let us return to our matrix-multiplication example to illustrate this concept. We have the following matrices

\[ A = \begin{pmatrix} 1 & -2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & -1 & 0 \\ 2 & 2 & 2 \\ 3 & 0 & 1 \end{pmatrix} \]

Say that the program has calculated the output \( C = A \times B \) incorrectly, and we wish to correct it. Now, we can easily write matrices of \( A \) and \( B \) as a sum of two matrices in an essentially random way. For example

\[ A = \begin{pmatrix} 3 & 4 & 0 \\ 2 & 1 & 3 \\ -1 & 0 & 1 \end{pmatrix} + \begin{pmatrix} -2 & -6 & 1 \\ -2 & 2 & 0 \\ 2 & 0 & -1 \end{pmatrix} \]

\[ B = \begin{pmatrix} -2 & 3 & 3 \\ -1 & 0 & 1 \\ 3 & 2 & 0 \end{pmatrix} + \begin{pmatrix} 2 & -4 & -3 \\ 3 & 2 & 1 \\ 0 & -2 & 1 \end{pmatrix} \]

We are just writing \( A \) as the sum of a random matrix \( R \) and \( A - R \), and similarly for \( B \). So, in the place of calculating \( A \times B \), we calculate the product of four entirely different matrix multiplications. Of course, this method multiplies run-time by a factor of 4. But, we assume that such correction is not often necessary. Also needed are matrix addition and subtraction, which we assume to be simple, quick, and reliable. The
multiplication becomes

\[
A \times B = \begin{pmatrix}
1 & -2 & 1 \\
0 & 3 & 3 \\
1 & 0 & 0
\end{pmatrix} \times \begin{pmatrix}
0 & -1 & 0 \\
2 & 2 & 2 \\
3 & 0 & 1
\end{pmatrix} = \begin{pmatrix}
3 & 4 & 0 \\
2 & 1 & 3 \\
-1 & 0 & 1
\end{pmatrix} \times \begin{pmatrix}
-2 & -6 & 1 \\
-2 & 2 & 0 \\
2 & 0 & -1
\end{pmatrix} \times \begin{pmatrix}
-2 & 3 & 3 \\
-1 & 0 & 1 \\
3 & 2 & 0
\end{pmatrix} \times \begin{pmatrix}
2 & -4 & -3 \\
0 & -2 & 1
\end{pmatrix}
\]

= \begin{pmatrix}
-1 & -5 & -3 \\
15 & 6 & 9 \\
0 & -1 & 0
\end{pmatrix}
\]

The corrector works as follows. Assume that, through random testing, we have previously determined that our program does matrix multiplication correctly for most input matrices: say, for all but at most a one in a million fraction of possible inputs. Now, each time we employ this self-correcting method, our problem of multiplying \(A\) by \(B\) is re-mapped to a problem of doing four matrix multiplications, each on a pair of matrices which are essentially random. Thus, for each of the four matrix multiplications, the multiplication will be correct, with chance of error at most one in a million. Hence, the chance that any of the four multiplications will be incorrect is at most four in a million. But this means that, for any \(A\) and \(B\), each time we employ this randomized correcting method, there is a very high probability of getting the correct value of \(A \times B\).

We conclude this section by discussing a simple checker and corrector for integer multiplication. The inputs are \(n\)-bit integers \(A\) and \(B\), and the product \(C\) is a 2\(n\)-bit number. We must decide whether or not \(A \times B = C\). Let us pick a small random integer \(r\), about \(\log n\) bits long. Now, calculate \((A \mod r)\) and \((B \mod r)\), multiply the resulting small numbers together, and again take the result \(\mod r\). To check if the result \(C\) is correct, compare this residue to \((C \mod r)\).

We will illustrate this method via a simple example. Consider a processor with 5-digit decimal representations. Asked to multiply 25736 by 36239, this processor claims that the exact result is 932646904. To check this multiplication, we pick \(r = 73\). So, \((A \mod r) = (25736 \mod 73) = 40\), and \((B \mod r) = (36239 \mod 73) = 31\). Then, we multiply these numbers together, and take the (result \(\mod r\)), obtaining \((40 \times 31 \mod 73) = 72\). We compare this residue with \((932646904 \mod 73)\) and find that it matches.

Why does this work? If \(A \times B = C\), then \(((A \mod r)(B \mod r) \mod r) = (AB \mod r)\) must equal \((C \mod r)\) for any value of \(r\). On the other hand, if \(A \times B \neq C\), it is possible to pick an unfortunate value of \(r\) for which the residues will still match. However, the probability (over the random choice of \(r\)) that this will happen is small. Thus, if we repeat this check a few times, we can readily make the chance of error as small as desired. This check is also computationally quicker and simpler than the original multiplication of \(A\) by \(B\). The check only requires that a pair of small numbers \(O(\log n)\) bits long rather than \(n\) be multiplied, and that four numbers be divided through by a small number \(r\).

When the checker detects a multiplication error, we can use a correction technique closely related to the matrix-multiplication corrector. Let us say that our chip fails on \(n\)-bit inputs \(A\) and \(B\). Then we pick two random numbers \(r_1\) and \(r_2\) from the set of all possible \(n\)-bit numbers, and calculate the values \(\frac{A + r_1}{2}\) and \(\frac{B + r_2}{2}\). Then, as a corrected value for \(AB\), we calculate

\[
4\left(\frac{A + r_1}{2}\right)\left(\frac{B + r_2}{2}\right) - 2r_1\left(\frac{B + r_2}{2}\right) - 2r_2\left(\frac{A + r_1}{2}\right) + r_1 r_2
\]

How does the correction work? Note that

\[
A \times B = (2 \times \frac{A + r_1}{2} - r_1) \times (2 \times \frac{B + r_2}{2} - r_2)
= 4\left(\frac{A + r_1}{2}\right)\left(\frac{B + r_2}{2}\right) - 2r_1\left(\frac{B + r_2}{2}\right) - 2r_2\left(\frac{A + r_1}{2}\right) + r_1 r_2
\]
How much work is needed to complete this calculation? The multiplications by 1/2 or 2 or 4 are easy on
binary numbers, and we assume addition and subtraction to be quick and reliable compared to multiplication.
Thus, the computation reduces to doing four multiplications. This is a significant cost, but, as usual, we
assume that it will only be needed in the rare case that the multipliers original output is incorrect.

1.3 Range Checks
We can use our knowledge of the software application to set acceptable bounds for the output. When setting
bounds on acceptance tests, we have to balance between sensitivity and specificity. Here, sensitivity is the
probability that the acceptance test catches an erroneous output (the coverage factor). It is the conditional

2 Single-Version Fault Tolerance
We first discuss software rejuvenation and then describe data diversity.

2.1 Software Aging and Rejuvenation
The topic discussed in this section is derived, in part from: V. Castelli et al., “Proactive Management of

Unplanned computer system outages are more likely to be the result of software failures than of hardware
failures. Software often exhibits an increasing failure rate over time, typically because of increasing and
unbounded resource consumption, data corruption, and numerical error accumulation. This constitutes a
phenomenon called software aging, and may be caused by errors in the application or operating system. Under
aging conditions, the state of the software degrades gradually with time, inevitably resulting in undesirable
consequences (e.g., the failure of the Patriot missile system, which resulted in loss of human lives, might
have been prevented had the operators heeded the advice that the system had to be restarted after every
eight hours of running time). Some typical causes of this degradation are memory bloating and leaking,
unterminated threads, unreleased file-locks, data corruption, storage-space fragmentation, and accumulation
of round-off errors.

To counteract software aging, a proactive technique called software rejuvenation involves stopping the run-
ing software occasionally, “cleaning” its internal state (e.g., garbage collection, flushing operating system
kernel tables, and re-initializing internal data structures) and restarting it. An extreme but well-known ex-
ample of rejuvenation is a system reboot. There are numerous examples in real-life systems where software
rejuvenation is being used. For example, it has been implemented in the real-time system collecting billing
data for most telephone exchanges in the United States.

Proactive fault management takes appropriate corrective action to prevent a failure before the system ex-
periences a fault. So, software rejuvenation is a specific form of proactive fault management which can be
performed at suitable times, such as when there is no load on the system, and thus typically results in less
downtime and cost than the reactive approach. Since proactive fault management incurs some overhead, an
important issue is to determine the optimal times to invoke it in operational software systems. Proactive
fault management can be greatly enhanced by the ability to predict the fault far enough in advance that
one can take action to avoid or mitigate its effects. Resource exhaustion by its very nature offers clues
that failure is imminent, in the form of parameters that can be monitored, extrapolated, and compared to
thresholds via suitable algorithms.

2.2 Data Diversity
Please read the following paper on data diversity, available from the course web site: P. E. Ammann and J.
2.3 Software Implemented Hardware Fault Tolerance

Read the discussion in the book.