CSCE 5760: Design or Fault Tolerance

Review
- Software Reliability and Fault Tolerance
- Testing
- Formal proofs and model checking
- Acceptance tests
- Wrapper

How to implement acceptance tests?
Use wrappers
The wrapper filters inputs and outputs

CSCE5760 Nov 05, 2019

CSCE 5760: Design or Fault Tolerance

Software Rejuvenation
- Periodically “update” software
- Examples given in book are not great
  - reboot, reclaim garbage, kill zombie tasks

- Periodic and Prediction based rejuvenation
  - Prediction: based on previous history of “bugs/errors”

- How often to rejuvenate
  - Cost of rejuvenation: Lost time
    - saving checkpointed software
  - Cost due to errors

- Security through software rejuvenation
  - Modular rejuvenation: rejuvenation one module at a time
Rejuvenation for Security

- The rejuvenation can be applied modularly and periodically in order to minimize the downtime of the system.
- Checkpoint good software module

We did not analyze the cost using prediction based rejuvenation
- can be a topic for term project

Another method for software fault tolerance
- data diversity

Think of dividing input space (domain values) into regions
  - inputs that do not cause errors
  - inputs that cause errors

Can we modify input – perturb inputs – to nudge the inputs away from error spaces

Again, practicality depends on the application domain
  - image processing or other statistical applications
Explicit vs implicit perturbation

Explicit - add a small deviation term to a selected subset of inputs
Implicit - gather inputs to program such that we can expect them to be slightly different

Example 1: software control of industrial process - inputs are pressure and temperature of boiler
Every second - \((p_i, t_i)\) measured - input to controller
Measurement in time i not much different from i-1

Implicit perturbation may consist of using \((p_{i-1}, t_{i-1})\) as an alternative to \((p_i, t_i)\)

If \((p, t)\) is in fault region - \((p_{i-1}, t_{i-1})\) may not be

Explicit perturbation

• Example 2: add floating-point numbers \(a, b, c\) - compute \(a+b\), and then add \(c\)
• \(a=2.2E+20, b=5, c=-2.2E+20\)

• Depending on precision used, \(a+b\) may be \(2.2E+20\) resulting in \(a+b+c=0\)
• Change order of inputs to \(a, c, b\) - then \(a+c=0\) and \(a+c+b=5\)

Software Implemented Hardware Fault Tolerance (SIHFT)

Can we overcome hardware failure with software support

Data diversity and replication

Think of transforming software to operate with original data and also data that is perturbed
Say for example, if we have software for a function \(f(a)\)
Write a new software that also implements \(f\), but uses input that is transformed – that is \(f(g(a))\)
Consider a simple transformation where the replicated software works on input that is multiplied by a constant $k$.

So the output of the replicated software should be $k$ times the results of original software.

The textbook shows how such an idea can detect stuck at faults.

If $i$th bit is stuck at zero, and if data transmitted has $i$th bit = zero; an error is not detected.

But how do we detect this? Send data and $2^d$ (shift).

If $i$th and $i+1$ bits are different, then you will notice an error.

What if $i$th and $i+1$ bits are both zeros?

Try $2^d$, $4^d$ etc.

May cause overflow.

Risk of overflow exists even for small values of $k$.

Even $k=-1$ can generate an overflow if original variable is equal to the largest negative integer that can be represented using two's complement (for a 32-bit integer this is $-2^{31}$).

Possible precautions:

Scaling up the type of integer used for that variable.

Performing range analysis to determine which variables must be scaled up to avoid overflows.

Use 2's complement.
What about if we are dealing with floating point numbers?

Some simple choices for $k$ no longer adequate

Multiplying by $k = -1$ - only the sign bit will change (assuming the IEEE standard representation of floating-point numbers)

Multiplying by $k = 2^l$ → only exponent field will change

Both significand and exponent field must be multiplied, possibly by two different values of $k$

To select value(s) of $k$ such that SIHFT will detect a large fraction of hardware faults – either simulation or fault-injection studies of the program must be performed for each $k$

Recomputing with Shifted Operands (RESO)

Similar to SIHFT - but hardware is modified

Each unit that executes either an arithmetic or a logic operation is modified

It first executes operation on original operands and then re-executes same operation on transformed operands

Same issues that exist for SIHFT exist for RESO

Transformations of operands are limited to simple shifts which correspond to $k = 2^l$ with an integer $l$

Avoiding an overflow is easier for RESO – the datapath can be extended to include extra bits
CSCE 5760: Design or Fault Tolerance

Recomputing with Shifted Operands (RESO) Example

An ALU modified to support the RESO technique

Example – addition

First step: The two original operands X and Y are added and the result Z stored in register

Second step: The two operands are shifted by \( l \) bit positions and then added

Third step: The result of second addition is shifted by same number of bit positions, but in opposite direction, and compared with contents of register, using checker circuit

CSCE5760 Nov 05, 2019

CSCE 5760: Design or Fault Tolerance

N version programming

N independent teams of programmers develop software

- Same specification or independent specifications
- Different develop tools, algorithms etc

If we assume failures are statistically independent, the probability that there are no more than \( m \) failures in \( N \) versions is given by

\[
p_{\text{ind}}(N, m, q) = \sum_{i=0}^{m} \binom{N}{i} q^i (1-q)^{N-i}
\]

Comparing results of different versions

- Exact match
- Approximate match

An example from textbook. A function of pressure and temperature, \( f(p,t) \), is calculated

- Action \( A_1 \) is taken if \( f(p,t) < C \)
- Action \( A_2 \) is taken if \( f(p,t) \geq C \)

CSCE5760 Nov 05, 2019
CSCE 5760: Design or Fault Tolerance

We have 3 versions computing some function of temperature and pressure of a system

Each version outputs action to be taken

Ideally all versions consistent - output same action

Versions are written independently - use different algorithms to compute \( f(p,t) \) - values will differ slightly

Example: \( C=1.0000; N=3 \)

All three versions operate correctly - output values: 0.9999, 0.9998, 1.0001

\( X_1, X_2 < C \) - recommended action is \( A_1 \)

\( X_3 > C \) - recommended action is \( A_2 \)

Not consistent although all versions are correct

We can consider using some tolerance and assume all of the above values are the same and equal to 1

---

Consistency Problem

**Theorem:** Any algorithm which guarantees that any two \( n \)-bit integers which differ by less than \( 2^k \) will be mapped to the same \( m \)-bit output (where \( m+k \leq n \)) must be the trivial algorithm that maps every input to the same number

\[ \Rightarrow \] that is we will ignore the \( k \) least significant bits

**Proof:**

We start with \( k=1 \)

0 and 1 differ by less than \( 2^1 \)

The algorithm will map both to the same number, say \( \alpha \)

Similarly, 1 and 2 differ by less than \( 2^2 \) so they will also be mapped to \( \alpha \)

Proceeding, we can show that 3,4,... will all be mapped by this algorithm to \( \alpha \)

Therefore this is the trivial algorithm that maps all integers to the same number, \( \alpha \)

**Exercise:** Show that a similar result holds for real numbers that differ even slightly from one another

---
Consensus comparison

If versions don’t agree - they may be faulty or not

Multiple failed versions can produce identical wrong outputs due to correlated fault - system will select wrong output

Can bypass the problem by having versions decide on a consensus value of the variable

Before checking if $X \geq C$, the versions agree on a value of $X$ to use

This adds the requirement: specify order of comparisons for multiple comparisons

Can reduce version diversity, increasing potential for correlated failures

Can also degrade performance - versions that complete early would have to wait

Confidence intervals

Each version calculates $|X-C| > \delta$ for some given $\delta$, version announces low confidence in its output

Voter gives lower weights to low confidence versions

Problem: if a functional version has $|X-C| > \delta$, high chance that this will also be true of other versions, whose outputs will be devalued by voter

The frequency of this problem arising, and length of time it lasts, depend on nature of application

In applications where calculation depends only on latest inputs and not on past values - consensus problem may occur infrequently and go away quickly
Correlation between versions
Need to make them independent
If not, our reliability formula does not hold

For example we have 3 versions and failure rate of a version is $q = 0.0001$

$$q^2 + 3q^2(1 - q) \approx 3 \times 10^{-8}$$

However, suppose 2 versions are correlated – that is if one of them fail both fail

Now we have basically 2 independent version

$$q^2 + 2q(1-q) = \text{approximately } 10^{-6}$$

If we cannot guarantee independence, then we need to change our formula for reliability to include conditional probabilities

Causes of correlation among N versions

- Common specification
- Intrinsically difficult problem
- Common algorithms
- Cultural factors
- Common software/hardware platforms

You can try to force independence

- Diverse specifications
- Diverse hardware and operating systems
- Diverse development tools and compilers
- Diverse programming languages
- Versions with differing capabilities
Recovery Block approach to fault tolerance

We can view this as N version programming with Primary and Spares

In general, the secondary versions need not be complete programs.

We can divide our algorithm into “blocks”, and test the output of the block with acceptance tests.

Then we only need alternatives to the “block” only

How to formulate the success probability of recovery blocks?

Assume that the recovery blocks are independent

\begin{align*}
E: & \text{ Event that output of a version is erroneous} \\
T: & \text{ Event that the acceptance test fails (that is we detected a failure of a version)} \\
f: & \text{ failure probability of a version (or } f= \text{ prob}(E)) \\
s: & \text{ test sensitivity, or } s= P(T|E) \text{ or test detected failure and there is an actual failure} \\
s: & \text{ test specificity } = P(\overline{E}|T) \text{ or there was an actual failure and test detected it} \\
n: & \text{ number of versions including primary} \\
\Pr(\text{success in stage } i) &= [P(T)]^{i-1} \cdot P(\text{No error and no test failure in ith stage})
\end{align*}

So, the probability that the n version recovery is successful is given by

\begin{align*}
\Pr(\text{Scheme is successful}) &= \sum_{i=1}^{n} [P(T)]^{i-1} P(\overline{E} \cap T)
\end{align*}

Note we need to find the probabilities of different event combinations
Example with $n=3$ and different values for $s$ and $\sigma$

Note that recovery block technique is an example of “Time Redundancy”
Distributed recovery block technique
Combining N version with recovery block technique
Consider 2 nodes (or two processors)
In one processor, we run the primary version with alternate as spare
In the second processor, we run the alternate as primary and primary as spare

![Diagram of nodes](image)

If node 1 is successful, use its output. If it fails, use the output from node 2.
Node 2 is also used if Node 1 does not produce output in finite time

Upon failure of Node 1, the roles of the nodes is switched; Node 2 will still continue to execute alternate version as primary
Can extend to more than 2 nodes

**Benefit?**

---

Exception handling
Another variation of previous examples
When a version produces an exception, invoke a special “exception handler”

Exception handler can be part of the software itself - can define hierarchically
Say a function f contains its own handler
Then if an exception is not handled by f, the calling function may handle it
If not next level up and so on

![Diagram of exception handling](image)

Function B has no handler for exception c
Exceptions can include acceptance tests
Or unexpected outcomes (like divide by zero, address violation, etc)

Some language allow you to define the exceptions
ADA
Java exceptions

What types of exceptions?
(a) domain or range failure
   Out of range inputs and outputs
(b) out-of-ordinary event(not failure) needing special attention
   End of File handling while reading a file
(c) timing failure
   missed deadline
(d) Resource failures

Graceful termination – don’t just core dump

Requirements of good exception handlers

(1) Should be easy to program and use
Be modular and separable from rest of software
Not be mixed with other lines of code in a routine - would be hard to understand, debug, and modify

(2) Exception-handling should not impose a substantial overhead on normal functioning of system
Exceptions be invoked only in exceptional circumstances
Exception-handling not inflict a burden in the usual case with no exception conditions

(3) Exception-handling must not compromise system state - not render it inconsistent
One final topic in software fault-tolerance.

**How do we model software failures?**
- Can we assume failure arrivals follow Poisson distribution?
- What distribution to use for the occurrence of software failures?
- How do we validate these failure distributions?

There have been many different models. Textbook discusses 3 models

The models different in how to obtain the failure rates.

---

Some assumptions

- Bug (defect) exists when the software is written
- Error occurs only when the program is executing and the bug is exposed
- Software does not have decay or wear related failures
- When a error is detected, the bug (actually could be multiple) is fixed and eliminated. But there could be other bugs in the software.
- Fixing bugs improves reliability

*In some cases, fixing a bug may introduce new bugs*

**Most reliability models are based on the idea that the failure rate at time t depends on how many bugs are still contained in the software**

Note in most simpler models, we assume that fixing a bug does not introduce new bugs
Jelinski-Moranda model

At time zero the software contains a fixed (finite) number of bugs $N(0)$

At time $t$, there are $N(t)$ bugs, usually $N(t) < N(0)$

The error rate is a function of $N(t)$

$$\lambda(t) = c \cdot N(t)$$

So, $\lambda(0) = c \cdot N(0)$

When a bug is discovered and fixed we have a new error rate (based on when the bug is detected).

Constant $c$ indicates a factor in terms of reliability improvement

Note as bugs are fixed, the error rate decreases and reliability improves

The time between successive errors is exponentially distributed and depends on $\lambda(t)$

So, before any error occurs the reliability of software or error free operation between $[0,t]$ is given by

$$R(t) = e^{-\lambda t}$$

Suppose an error occurred at time $\tau$, the reliability of the system for next $t$ units of time, or error free operation from $[\tau, \tau + t]$ is given by

$$R(t \mid \tau) = e^{-\lambda(\tau+t)}$$

As more bugs are fixed, the error rate $\lambda(t)$ reduces and thus the reliability increases.

This model assumes that all bugs contribute equally to error rate; the constant $c$ describes this contribution.

In reality, some bugs may be hidden and will not be discovered for a long time; some bugs are more serious than others.
Littlewood – Verrall model

Assumptions. Initially we have $N(0)$ bugs in the software
At time $t$, there are $N(t)$ bugs still in the software
So, $M(t) = N(0) - N(t)$ is the number of bugs detected $[0,t]$

The error distribution is a non homogenous Poisson distribution with $\lambda(t)$ but unlike normal Poisson distribution with a fixed value, the rate itself is considered a random variable and depends on a Gamma distribution. Gamma has two variables $\lambda$ and $\psi$

The reliability using this model is:

$$R(t) = \left( 1 + \frac{t}{\psi(0)} \right)^{-\alpha}$$
$$R(t | \tau) = \left( 1 + \frac{t}{\psi(M(\tau))} \right)^{-\alpha}$$

The first is reliability in $[0,t]$ while the second is the conditional reliability that an error occurred at $t$ and what is the reliability in $[\tau, t]$
The third model described in textbook is by Musa and Okumoto

In this model, we even include the impact of testing software on software reliability.

We assume that with more testing, there will be fewer bugs remaining when the software is released and thus the reliability of the software improves with longer testing.

As with Littelwood model, M(t) is the number of bugs still in the software.

The model assumes that the error rate (\( \lambda(t) \)) after testing the software for \( t \) time units is given by

\[ \lambda(t) = \lambda_0 e^{-c \mu(t)} \]

\[ \mu(t) = E(M(t)) \]

Is the expected number or errors occurring in [0,t] testing.

But how to find this value?

\[ \lambda(t) = \frac{\ln(\lambda_0 ct + 1)}{c} \]

\[ \lambda(t) = \frac{\lambda_0}{\lambda_0 ct + 1} \]

\[ R(t) = e^{-\int_0^t \lambda(z)dz} = e^{-\mu(t)} = (1 + \lambda_0 ct)^{-\frac{1}{c}} \]

\[ R(t | \tau) = e^{-\int_{\tau}^{t+\tau} \lambda(z)dz} = e^{-[(\mu(\tau+t)-\mu(\tau))]} = \left(1 + \frac{\lambda_0 ct}{1 + \lambda_0 ct \tau}\right)^{-\frac{1}{c}} \]
The left graph is for $c=1$ and different $\lambda$. The right graph is for $\lambda=1$ and different $c$’s.

In both graphs, we notice that the failure rate (error rate) decreases with testing. So, the longer you test, the more reliable the software.

However, the decay in error rate is slow and thus this model indicates a LONG testing time leads to “diminishing returns.”

Other models: does fixing a bug introduce new bugs?

We can use a distribution to describe the failure rates of these newly introduced failures. But how many does each bug fix introduce?

We can define an expected number of bugs introduced:
- this can be based on our experience during testing
- or use a monotonic growth model and Gamma distributions

Note reliability increases as more bugs are detected and fixed.
- $So$, fixing later bugs may introduce more bugs!

These models must be validated with actual data collected.
- AT&T, NASA and a few others have provided data that could be used.
- Note it is difficult to collect data on bugs
- bug fixes, nature of the bug (specification, design..) etc.
Consider problem 5.4 on page 187.

Here we use a Bayes model for errors.

The probability of uncovering a bug after testing for $t$ time units is

$$1 - e^{-\mu t}$$

Let $q$ be probability that the software has at least one bug or the probability that the software is bug free $= p = 1 - q$

Suppose after testing the software for some time, $t$, no bugs were discovered.

Your probability that the software is bug free need to be adjusted

Using Bayes formula, we need find the probability that the system is bug free given that no bugs were found after testing for $t$ time units.

$$\text{Prob}(A|B) = \frac{p}{p + qe^{-\mu t}}$$

$A$: software is bug free

$B$: no bugs discovered after testing for $t$ time unit

Even though software reliability is controversial, we still want to make sure that systems software and critical software are reliable

- Reliable kernel functions
- Library functions
- Communication libraries

Need to design into the code the capability to detect exceptions (acceptance tests) and exception handling

Chapter 6 deals with checkpointing and recovery

The idea is straightforward. Save partial results (and state)

If a failure, rollback to previously saved state and continue

This is an example of time redundancy – may lose some computations.