

CS 350: Introduction to Software Engineering

Slide Set 3
C. M. Overstreet
Old Dominion University
Spring 2005

Why GCS (Guidance and Control Software)?

- NASA & FAA interest in “ultra-reliable” software
- NASA/Langley lead in previous project (still core of knowledge about project)
- Complex but not too complex
- I like it because:
 - Real-time system
 - Application knowledge required (just like real life)
 - Really messy in places (just like real life)
 - ~100 pg. req. doc.
 - Bad features (just like real life)
 - I'd do a bunch of things differently but we're stuck with other people's bad decisions

Comments on Real Time

- In a real-time system:
 - Computational tasks **must** complete by a deadline
 - Often cyclical: computations repeat at a specified frequency
- Operating system considerations
 - Many OS's allow you to schedule the **start** time for a task (see cron in UNIX).
 - Much harder to guarantee that a task will **complete** by a deadline, particularly for interrupt driven systems

Software Experimentation

- Std Problem is SE: many opinions, few facts
- Purpose of GCS: get some real data
- Study a software systems that:
 - Doesn't cost too much to build
 - Small but not too small
 - Is in the "right" application domain
 - Flight control, real time
 - Can be used for data collection
 - Want failure rate data (e.g. MTTF)
 - Want to understand kinds of errors people make

N-version Software

- Purpose: increase software reliability
- One idea:
 - Develop one requirements document
 - Give copies to several different development groups (say 5)
 - Each group develops complete system
 - Independently! No info exchange!!
 - Run all five systems:
 - Give each identical input
 - Wait for each to produce required output
 - Compare 5 outputs & take majority as the correct output
- Question: does this work?

Failure Probabilities - 1

- In software reliability world, failures are treated as "random" events (even though they aren't)
- Each programs has "input space":
 - The set of data the program will receive when running, including the fact that it will see some inputs frequently, others rarely.

Failure Probabilities - 2

- Then probability of program failure is based on idea:
 - Randomly pick a data value from the program's input space (where selecting particular values reflects how frequently they will be seen when the program really runs).
 - Observe whether or not the program fails
 - Do this for all values in input space (properly weighted).
- $\text{Pr}(\text{failure with a randomly selected input}) = \frac{\text{number failures}}{\text{number tests}}$

Failure Probability Comments

- Simple concept. But involves several potentially hard problems:
 - What is the distribution of inputs the program will really encounter in use?
 - Input space often too large to run all program with all possible inputs
 - How do you tell when the program fails (assuming it doesn't crash)?
- Typical industry interest is how often software will fail when used by their customers. This is a similar idea.
 - Depends on both frequency of use and typical input data

N-version Software - 2

- Statistical concept: **event independence**
 - Knowing one event occurred doesn't change probably of another event.
 - Independence $\Leftrightarrow \text{Pr}(A \wedge B) = \text{Pr}(A) \times \text{Pr}(B)$
 - Often, this is not true
- Key Question:
 - Do separately developed versions really fail independently?
 - Or if one fails, does that make it more likely that another will fail also?

Oracle Problem - 1

- When testing software, how do you detect all erroneous outputs?
 - Unsolved SE problem for many applications

Oracle problem - 2

- Federal Aviation Administration (FAA) specifies that the failure rate for commercial aircraft of less than 10^{-9} .
- How many test cases should you run to determine this?
 - Generally considered a statistical problem: if the software is given a "random" input, what is the likelihood of the program producing incorrect output (assuming it doesn't crash)?
- How do you check the answers?

GCS Goal

Generate data for software reliability study

- What types of mistakes do programmers make?
- How often do different programmers make the same mistake?
- If "approved" software development procedures are used (here, RTCA/DO-178B--required by FAA), how reliable will the software be?

Estimates, program 2

- New estimates:
 - How long in each phase
- Estimation procedure
 1. Understand assignment
 2. Based on your experience (from prog 1?), estimate LOC for prog 2; modify if appropriate
 3. Use your productivity data from prog 1 (min/loc); modify if appropriate
 4. Time est. = size (from step2) X (min/loc from step 3)
 5. Adjust time estimate, if appropriate
 6. Use phase percentages from prog 1 to estimate phase percentage (then minutes) for prog 2. Again, adjust, if appropriate

Form example

- Minutes/LOC, LOC/Hour
 - From prog 1, modified as appropriate
- Time in phase (min) (based on prog 1, perhaps modified)
 - Planning
 - if you spent 15% on prog planning, start with 0.15*time est prog 2
 - Code Design—same again
 - Test Design
 - Code
 - Compile
 - Postmortem
 - Total Time
 - Max Time
 - Min time

GCS Context - 2

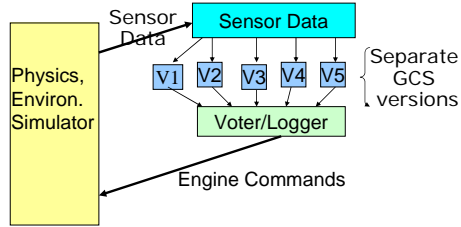
Use n-versions to address the oracle problem:

- Assume, say, 5 complete versions of GCS are available.
- Give each a set of inputs.
- After all finish, compare outputs.
- If identical repeat (run another test)
- If at least one differs
 - We've found an error
 - Don't know which program.
 - May be a problem with ambiguous requirements.

Does this solve the oracle problem?

GCS Context - 3

Execution context:



Bigger Context, Design

- Viking project was large
- Done by teams, different teams for different phases
- Main phases:
 - Launch rocket (with Viking lander as payload)
 - Fly payload to Mars, place into orbit
 - Start decent of payload to target location; parachute deployed
 - **Control descent to surface of Mars**
 - Conduct scientific mission & transmit data to Earth
 - Analyze data collected

Our part!

Reading the spec - 1

- Read pp. 3-18, 89-123 for Mod.
 - Read as reference doc. That is, most important is to be aware of what is there and where there rather than to memorize lots of details.
 - Point: when you get to code design, you will (we hope) remember that something about this was discussed in several places, then reread those parts.

Reading the spec - 2

- Examples:
 - Section on vector notation and frames of reference: if you design, code, or test a module that deals with frames of reference, you will need to understand this discussion
 - Rotating history variables: likewise.
 - Point: wherever certain variables are modified, it's up to you to remember that some variables maintain a history of previous values. This must be in the design, the code, and the test data.

Reading the spec - 3

- Make sure you understand what the intro says about control variables.
- Range checking: make sure you understand when and for what variables this is to be done.
- Make sure you understand the implications of "number representations" and "conversion of units" (pg. 14)

Format of GCS Soft Req. Doc

- Data Flow Diagrams
 - Context diagram
 - Data flow (solid edges)
 - Control flow (dashed edges)
 - Processes (bubbles)
 - Data store (boxes)
 - Levels
- Data element dictionary

Data Element Dictionary

For each variable which two programmers (modules) may share, include:

- Name
- Short description
- Where used
- Units
- Range (for enumerated types, meaning of each value)
- Type (real, integer, logical, array)
- Use (data, control, condition)
- Data store location
- Accuracy

Quiz

1. In what modules should the value of the variable `atmospheric_temp` be checked to see if it is in range?
2. In what modules should `ae_switch` be range-checked?
3. What modules use the variable `ae_status`?
4. What modules change the value of `ae_status`?
5. What is the value of `ae_switch` if the axial engines are off?
6. Who calls `gcs_sim_rendezvous`?

Viking Lander



Viking Orbiter



GCS System

- See NASA doc for other pictures
- Physical systems on lander:
 - Engines
 - 3 axial engines (on bottom)
 - 3 pairs of roll engines (around sides)
 - Communications

Physical Systems (cont.)

- Sensors
 - 3 accelerometers (x, y, z directions)
 - Altimeter radar
 - Doppler radar
 - 3 gyroscopes (x, y, z directions)
 - 2 temperature sensors
 - Touch down sensor

Physical Systems (cont.)

- Misc.
 - Communication hardware
 - GCS_SIM (not really needed for study): a simulator for environment, physics of spacecraft
 - Parachute release switch

GCS_SIM_RENDEZVOUS Purpose

- Problems
 - Keep versions of code from diverging over time from numerical rounding errors
 - Keep testing going even when some code aborts
 - Support error detection (the oracle problem)
- Remember:
 - Pretend purpose: fly spacecraft
 - Real purpose: study effectiveness of n-version software

NASA Testing Architecture:

- Use simulator for testing:
 - Simulator provides basic feedback loop
 - Sets sensor values (radar, acceleration, temp., etc)
 - GCS reads sensor values and decides which commands to send to engines
 - Simulator reads those engine commands, simulates their effect on the spacecraft sensors in the Martian environment, and sets sensors to those values
 - Loop till GCS shuts down system

Testing arch (cont.)

- Back to n-version objective
- NASA test harness consists of simulator and test driver
 - Can handle 20 different versions:
 - Load data (20 copies) for all versions
 - Start each version as separate process
 - Wait a specified amount of time for them to "rendezvous"
 - Checks all outputs, log differences (a program fault!)
 - Run simulator using outputs from version 1
 - Reload data (again 20 copies) for all versions
 - Repeat until touchdown sensed.

Testing arch. (cont.)

- How does this approach solve rounding error problem?
- How does this approach solve version abort (or even inf. loop) problem?
- How does this approach address oracle problem?

Specification Intent

- Describe software which:
 - Could actually be installed in and fly a lander without any source code changes
 - Can be run with an environment/physics simulator without any source code changes
 - Can be run in an n-version experiment without any source code changes
- If you do your code right, all three uses should be possible without any code modifications

Some Notations

- PP. 11 ff. explain notations used for arithmetic operations
 - Scalar multiply (.)
 - Matrix multiply (x)
 - Term-wise multiply (*)
- Use of history values
 - E.g., GP_VELOCITY(1:3,0:4)
 - Needed for numeric integration
 - When used with arithmetic expressions, ignore history--use current value

Processes (typical real-time sys)

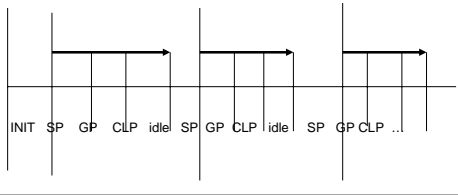
- Phase 1 - sensor processing
 - ASP (accelerometer)
 - GSP (gyroscope)
 - TSP (temperature)
 - ARSP (altimeter radar)
 - TDLRSP (radar 2)
 - TDSP (touch down)
 - CP (report)

Process (cont.)

- Phase 2 - guidance processing
 - GP (determine velocity, acceleration, altitude needed for phase 3)
 - CP (report)
- Phase 3 - control law processing
 - AECLP (axial engines)
 - RECLP (roll engines)
 - CRCP (chute release)
 - CP (report)

Timing

Typical Time Line



Process is restarted (by HW clock) every 1/150 sec

A Little Bit of FORTRAN

- Data Stores in FORTRAN are represented by COMMON blocks, in C++ by structures

- Data types:

FORTRAN	C++
real*8	double
integer*2	short
integer*4	long
logical*1	char

- FORMAT statements specify arrangement of output:
e.g.

- i4, means use 4 positions to write the value of an int
- a6, use 6 positions for a string
- e23.14, use 23 positions for a float, with 14 after the decimal

Level 0 Spec (context diagram)

- How does system communicate with outside world

- Reads sensor gadgets
- Syncs with GCS_SIM (through calls to gcs_sim_rendezvous at the beginning of each subphase)
- Controls rockets, chute, sends data

Level 1 Spec

- INIT_GCS loads (not your job!)
 - Guidance State
 - Run Parameters
- RUN_GCS (is what you're coding)
 - Run_gcs calls all of the modules.

Level 2 Spec

- Details of Init & Run

Level 3 Spec Format

- Look at cp as example.
 - Purpose: general short overview
 - Construct and send new data packet
 - What's sent varies depending on subframe
 - Inputs: what's used
 - Many variables
 - Output: what's changed
 - Data packet, status
 - Then as much English text, equations, pictures, tables (well, maybe not!) to describe required functionality.

cp process steps

- Set status
- Build data packet
 - Synch pattern
 - Determine sequence number
 - Build sample mask
 - Build data section
 - Calculate checksum

Assignment

- Read ch. 4 & 5 of psp text.
- A weekly activity summary and a job number log will be due at the end of next week.
 - This is based on a time recording log, so be sure you are keeping this up to date.

Programming Assignment

1. Implement the cp module.
2. Create test data for cp.
This may take more time than coding!
3. Show your code works by checking it with your test data.
4. Show your code works by checking it with other people's test data.
5. Show your test data works by checking other people's code with it (and your test driver)

Eventual Project

- Take a few modules you have developed individually (while you're learning PSP)
- Form a project team
- Team will write and integrate remaining modules (while you're learning TSP)

Other requirements

- What have I likely missed in this approach?
 - Need to realize that the requirements that apply to all modules are described in the front of the document rather than repeated in each module.
 - Some of the text requires a lot of thinking and the derivation of some mathematics before you can design any code.
 - The length of text may not predict how much code is needed.
 - The length of text may not predict how much time is needed.
