

System Engineering and Testing Strategies

RSS Lecture 7
Wednesday, 24 February 2010
Prof. Seth Teller

My Goals Today

- Discuss system engineering from an intellectual and practical standpoint
- Introduce a "toolkit" of ideas and techniques that you can adopt in your own engineering endeavors
- Get you thinking about your own useful engineering practices

Caveat Auscultator (Listener beware)

- Some of this material will be new to you; some will be familiar
 - It doesn't hurt to hear things twice.
- Some things you will probably agree with; some things you probably won't
 - But surely you're used to this by now.

Process View

- Engineering is a Means ...
 - Specifying: describing *what* to make
 - Designing: describing *how* to make it
 - Implementing: *realizing* actual artifact
 - Validating: convincing yourself (and others) that artifact *works* as specified
- ... to an End
 - Namely: an artifact with desired *behavior*

Human View

- Engineers are people who:
 - *Conceive of* and *execute* ways to optimize an underspecified tradeoff between possibly conflicting goals (such as performance, cost, etc.) ...
- ... subject to *physical* constraints:
 - Natural: Laws of physics, i.e., reality
- ... and to *social* constraints:
 - Cultural: Law, morality, ethics ...

Conception & Execution

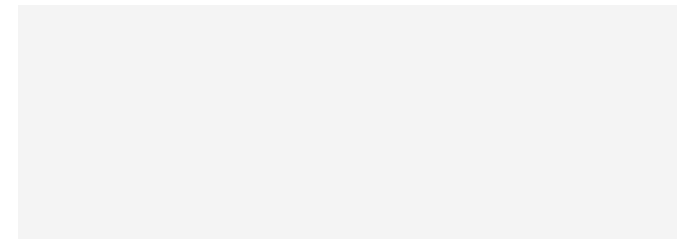
- Conception:
 - A *mental model* of artifact, constraints, and assumptions about environment
- Execution:
 - Putting the mental model into practice
 - Observing whether it *predicts* behavior under real-world conditions (and whether environmental assumptions are justified)

Essence of Engineering ...

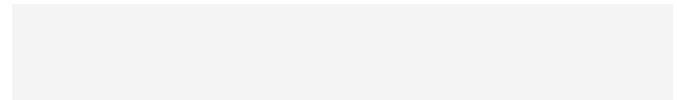
- ... Process is the (typically iterative)
 - *Formation* of a mental model;
 - *Implementation* of prototype artifact; and
 - *Observation* of its behavior, leading to:
 - Revision of designer's operative mental model
 - Revision of current design or implementation
 - (Or both)
- ... Until desired behavior is achieved

Consequences of Anomalies

- If it "looks wrong" to you, two possibilities:



- If things "look wrong," it's an opportunity to



... And if it looks correct?

- Is it correct?
- Sure, it often is correct. But that doesn't mean that it *always* is or *has to be* correct!
- Can boil these ideas down to an aphorism:
 - "Don't sweep anomalies under the rug."
 - In other words, anomalous behavior presents a great opportunity to learn something!

Documentation: JavaDocs

- JavaDocs comprise:
 - Declarations
 - Comments } for some code corpus
- Can help match mental models, but...
- ... teammates' agreement to make the code implement the *intent* stated in the comments essentially amounts to a *social contract*

A Concrete Strategy

- Iterative Prediction, Test, Evaluation
- Not:
 - "Hmm, now that I have modified this element, let's see what happens"
- Instead:
 - *Predict* outcome of some well-defined test
 - *Perform* the test
 - *Evaluate* actual outcome; form conclusions
 - Simple, systematic approach

Team Mental Models

- This strategy can be pursued by an individual, or by an entire team
- Also useful for resolving discrepancies in mental models *within* a team
- How?
- Inexhaustible source of experiments

Self-Checking Code

- Idea: make machine work for you
- For each algorithm/module, write a “checker” that inspects its *output* for the properties that it should have
- ... same idea applies to module *input!*
 - Postconditions (A) == Preconditions (B)

Pre/Postconditions, Invariants

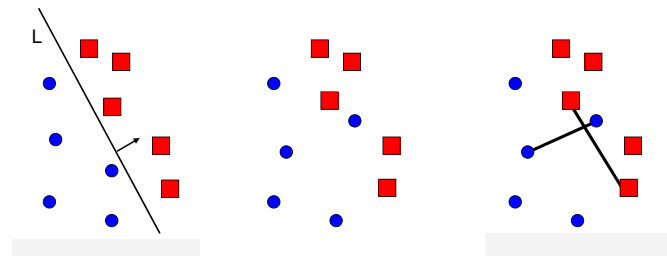
- Preconditions, postconditions and invariants are commonly used in “design-by-contract” engineering.
- Precondition - what must be true when a method is invoked. When a precondition fails, the fault lies .
- Postcondition - what must be true after a method completes successfully. Provided that the precondition was met, when a postcondition fails, the fault lies .
- Class Invariant - what must be true about each instance of a class after every method call (including construction!). When a class invariant fails, fault could lie in the , in the .
- Another common kind of invariant is internal – any condition(s) in the implementation which we know must always hold.

Teammate-Checking Code

- Twist: for each module you write, ask a *teammate* to write the checker (could be as fine as function grain)
- Multiple benefits:
 - Validates your solution (as before)
 - Decreases chance that checker succeeds due to an invalid *assumption* (why?)
 - Facilitates agreement of your mental model with your teammate’s model
 - Exploits a natural human characteristic: *competitiveness* (s/he acts as *adversary*)

Witnesses: “Prove it!”

- Example: linear separability (LP)
 - Given point sets $\{A_i\}$, $\{B_i\}$, i in $[1..N]$
 - Identify line L s.t. all A_i lie above L & all B_i lie below L , or show that no such L exists



Caution: A Practical Issue

- Make sure your checking, reporting, witness etc. code has no *side effects* that enable correct algorithm function
- Otherwise, when you disable your self-testing code, bugs may emerge
- Examples?

Adversary

- Someone/something that tries to
 - Find holes in your correctness argument (e.g. as A did for R & S of RSA security)
 - Produce *inputs* that break your code (e.g., by violating your assumptions)
 - Produce *conditions* that break system (more than just program's *formal input*)
- Adversary can be a [redacted], [redacted], or even a [redacted]

Some Adversarial Strategies

- Generate challenging *inputs* ...
 - Exhaustively
 - Randomly
 - Qualitatively
 - Deviously (e.g., provoke a teammate to do it)
- ... and nominal or anomalous *conditions*:
 - Notional environment, arranged to expectations
 - Missing or mis-wired connectors
 - Misbehaving sensors
 - Depressed all-stop buttons
 - Undefined environment variables
 - Misconfigured networks, remote hosts, etc.

Self-Checking Summary

- Pit each module against itself.
- Aphorism: “Make each module prove itself before you trust it.”

Test Harness

- Battery of test cases applied to a system to validate its responses
- We've seen these in "software only" systems, with "soft-copy only" inputs
- But what about robotics? How can we validate sensors and actuators using only software?

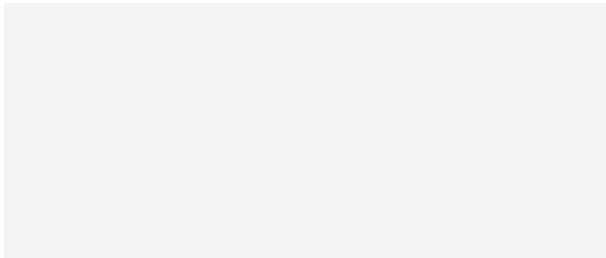
Robotics is Different!

- Robots are subject to "hard state" fundamentally not under s/w control
- Consider relation of proprioceptive (e.g., odometry, IMU) and exteroceptive (e.g., vision, ranging) sensor data for motion
- Actuators pose analogous problems
- Simulation can be useful*, but ...
- Real world is the *only way* to enforce absolute consistency of env't, state

*Rod Brooks: "Simulation is doomed to succeed." What does that mean?

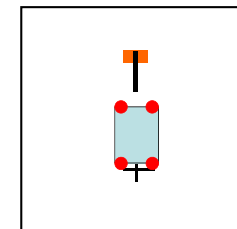
Example

- Bot commands forward motion, but sensed wall ahead isn't getting closer!
- Many possible explanations:



Robotics Test Harness

- Place robot in a *known* environment ... thus actions have known outcomes
- For concreteness, imagine harness for:
 - Odometry
 - Motor drivers
 - Bump sensors
 - Visual servoing
 - Arm driver
 - Gripper sense

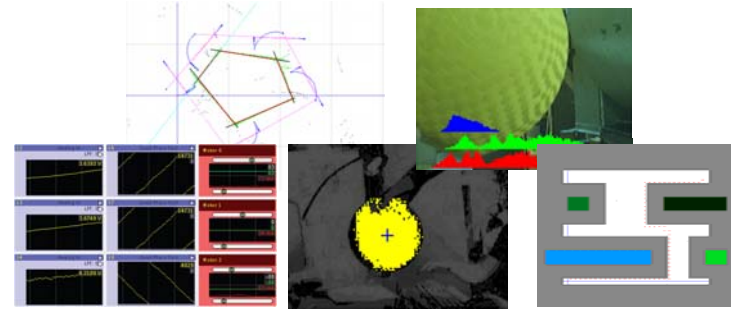


Self-Checking Summary (cont.)

- Pit system against known environment.
- Aphorism (Feynman):
"You can't fool Mother Nature."

Transparency of Live State

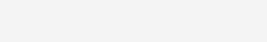
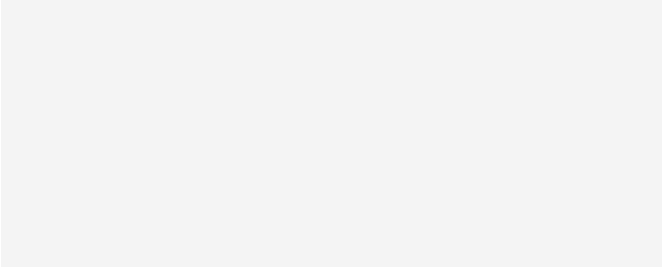
- Make live system state graphically *visible* (at least while debugging)
 - Generalizes print statements (& more fun)



Benefits of State Visualization

- Exposes otherwise hidden system state
- Exploits high-bandwidth visual system
- Speeds iterative development cycle
- Increases achievable complexity
- Useful for communicating results
 - To teammates (for matching models)
 - To others (for demos, presentations...)

Hierarchical Testing

- Idea underlying all CS: 
- This suggests a *recursive* test strategy: 

Longitudinal Testing

- Running over long time scales, spatial excursions may expose *vulnerabilities*:
 - Memory leaks, desynchronization, insufficient buffering, drift, decalibration...
- Longer runs increase the likelihood of encountering “good” conditions/inputs
- Course challenge requires repeated runs of 10-15 minutes (good practice!)

Consider Pair Programming

- Treat programming as an actual *collaborative* activity among peers
- One person types, the other person constructively comments, questions
- Trade roles at agreed-upon intervals
- Prompts useful design discussions
- Shortens design iteration dramatically
- *Try it!*

General Comments

- You’ve heard it all before
 - “Think before you code”
- My variation on this:
 - “Validate as you design and implement”
- Tangible benefits in rapidity of prototyping & achievable complexity while retaining confidence in correctness

Summary

- Engineering is about **predictive power**
- Primacy of **mental models** in testing
 - Both individual and shared
- Importance of **transparent state**
- Strategies for **iterative design & test**
- Potential of **adversarial self-checking**