Using Simplicity to Control Complexity

Paper by Lui Sha (UIUC)
Presented by Foo

Does diversity in construction improve robustness?

- Author investigates relationship between complexity, reliability, and development resources
- Presents approach to building a system that can manage upgrades and repair itself when complex software components fail

Introduction

- He shows dividing resources for diversity can lead to either improved or reduced reliability (duh!), depending on the architecture
- Key to improving reliability is not the degree of diversity, but the existence of a simple and reliable core component that ensures system's critical functions
- This is what he calls *using simplicity to control complexity*

Relationship between reliability, development effort, & software logical complexity

- Computational complexity – number of steps to complete
- Logical complexity – number of steps to verify correctness (function of number of cases or states that testing process must handle)
- Residual logical complexity – reusing component => no contributed logical complexity

Three Postulates

- P1: Complexity breeds bugs
  - Reliability ? complexity ?
- P2: All bugs are not equal
  - For given degree of complexity, reliability function has monotonically decreasing rate with respect to development effort
- P3: All budgets are finite
  - Diversity is not free (need to divide available effort)

Simple model that satisfies postulates

- Exponential reliability function

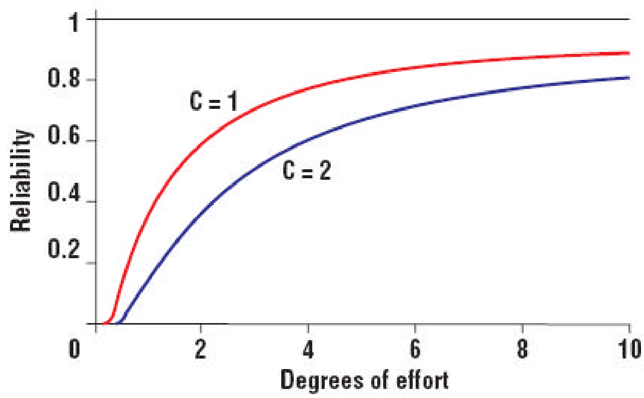
$$R(t) = e^{-l\,t}$$

$$l \propto \text{complexity } C$$

$$l \propto \text{development effort } E$$

$$\therefore R(t) = e^{-kCt/E}$$

Example



Two Examples to illustrate model

- Two well-known software fault tolerance methods that use diversity
  - N-version programming
  - Recovery block
- For fairness, assume
  - Faults independent under N-version programming
  - Acceptance test perfect under recovery block
- He states these assumptions are, in practice, not easy to realize

N-version programming

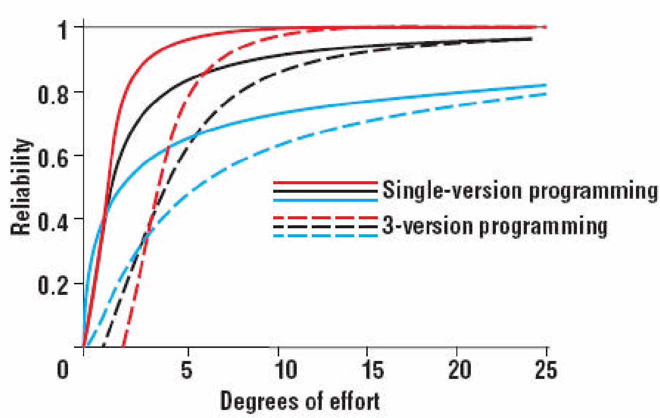
- Assume C=1
- When N=3,

$$R_3 = R_{E/3}^3 + 3R_{E/3}^2(1 - R_{E/3})$$

$$R_{E/3} = e^{-3/E}$$

- Assumption that we divide a team into 3 groups, so effort contributed per group is 1/3 the usual team effort

Under that assumption, single-version programming performs better nearly all the time



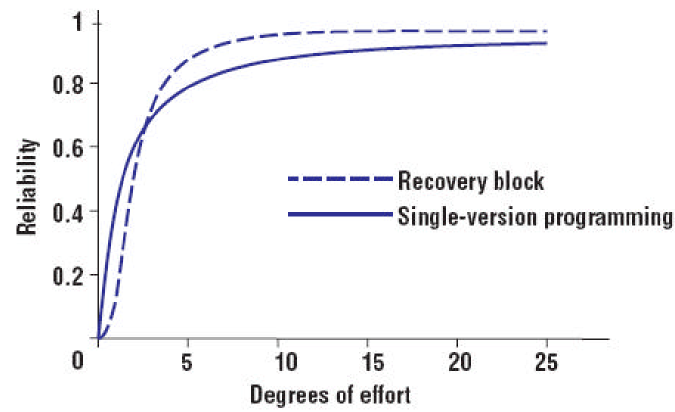
N-version programming

- Even if assumption is wrong, the other issues are
  - No method to assure faults in different versions are independent
  - No reliable method to quantify impact of potentially correlated faults
- FAA DO 178B discourages use of N-version programming (Probably some Federal Aviation Administration guidelines for aircraft/airport software design)

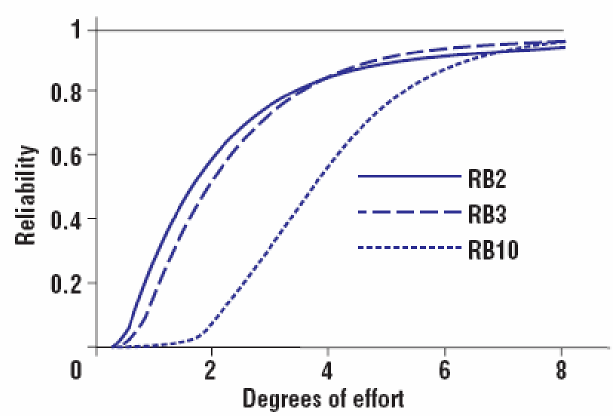
Recovery block

- Like N-version programming, but exist an acceptance test, so no majority voting required

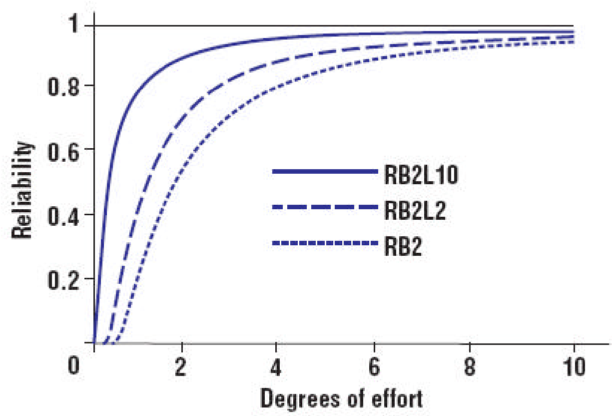
$$R_B = 1 - (1 - R_{E/3})^3$$



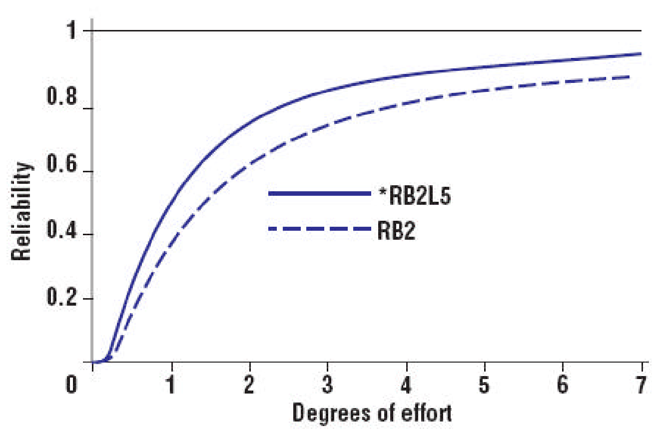
Varying number of alternatives



Varying complexity of second block



Imperfect acceptance test for RB2L5 (i.e. reliability is 0.2)



Conceptual framework

- Using simplicity to control complexity lets us separate critical requirements from desirable properties
- Example:
  - In sorting, critical requirement is to correctly sort, desirable property is to sort them fast
  - Suppose bubble sort can be verified but not quick sort
  - Solution is to use quick sort, then pass output to bubble sort, thus obtaining same computational complexity
- Can exploit features and performance of complex software even it cannot be verified, provided it is possible to guarantee the critical requirements with simple software

Forward recovery solution

- Now he shows how to apply this idea in the context of automatic control systems (ACS)
- A difference (error) often exists between actual device state and set-point (desired state)
- Feedback control is itself a form of forward recovery
- In this framework, incorrect control-software outputs translate to actuation errors
- So must contain impact resulting from incorrect output and keep system within operational constraints

Example

- One way to do that, is to keep device states in an envelope established by simple and reliable controller
- Example
  - Boeing 777 flight control system uses triple-triple redundancy (3 parallel control channels, each channel has 3 different processors i.e. Intel, Motorola, AMD)
  - Application-software level has 2 controllers, new sophisticated primary controller, and old 747-based secondary controller

Forward recovery

- Using forward recovery in software systems is an exception rather than the rule due to perceived difficulties
- He discovered we can systemically design and implement forward recovery for ACS if the system is piecewise linearizable (due to recent advancement in linear matrix inequality)

Simplex architecture

- Control system divided into high-assurance-control subsystem (HAC) and high-performance-control subsystem (HPC)

- HAC's simple construction allows one to leverage power of formal methods and rigorous development process
- HPC complements HAC, can use more complex and advanced control technology, same rigorous standard must also apply

HAC uses the following technologies

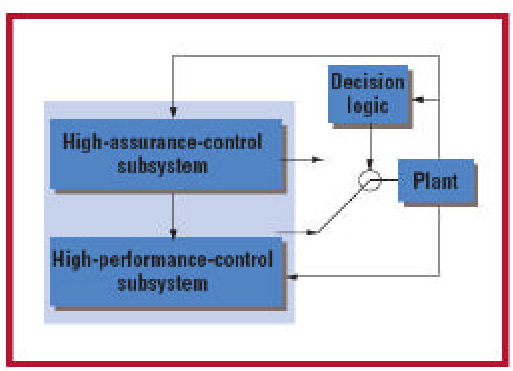
- Application level
  - Well-understood classical controllers designed to maximize stability envelope
  - Trade performance for stability and simplicity
- System software level
  - High-assurance no-frills OS kernels
  - Trade usability for reliability
- Hardware level
  - Well-established and simple fault-tolerant hardware configurations
  - E.g. pair-pair or triplicate modular redundancy
- System development and maintenance process
  - High-assurance process appropriate for application
- Requirement management
  - Limits requirements to critical functions and essential services

Pursuing advanced control tech. in HPC

- Application level
  - Include those difficult to verify e.g. neural nets
- System software level
  - COTS real-time OS and middleware designed to simplify app. Development
  - Dynamic real-time component-replacement capability in middleware layer
- Hardware level
  - Standard industrial hardware
- System development and maintenance process
  - Standard industrial software development processes
- Requirement management
  - Handle requirements for features and performance here

Figure for the less imaginative

- Normally HPC controls plant
- Decision logic ensures plant's state under HPC stays within HAC-established stability envelope
- Otherwise HAC takes control



Non-safety critical systems

- Some real-time control applications such as manufacturing systems not safety critical
- Need high degree of availability
- Main concern is application-software upgradability and availability
- Can run Simplex architecture middleware on top of standard industrial hardware and real-time OS

Educational purposes

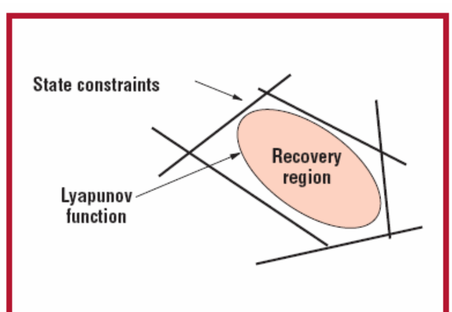
- He made his group at UIUC develop web-based control lab called *Telelab*
- Uses physical inverted pendulum
- You can submit software through the web and replace existing control software without stopping normal control
- Can watch how software improves control through steaming video
- Can embed bugs and let Telelab detect deterioration of control performance and take back control, keeping pendulum from falling down
- Demonstrates feasibility of building systems that manage upgrades and self-repair

Details

- In plant (or vehicle) operation, set of state constraints, called *operation constraints*, represent devices' physical limitations and safety, environmental, and other operational requirements
- Represent them as normalized polytope (n-dim figure whose faces are hyperplanes) in the system's n-dim state space

2-D Example

- Each line represents constraint, e.g. engine rotation must be no greater than *k* rpm
- States inside polytope are *admissible states*
- Must ensure system states are always admissible



Requirements

- Able to remove control from faulty control subsystem and give it to HAC subsystem before system state becomes inadmissible
- HAC subsystem can control system after switch
- System state's future trajectory after switch will stay within set of admissible states and converge to the set-point

Switching rule

- Cannot use polytope's boundary as switching rule, just as cannot prevent collision by stopping a car just before it hits wall
- Subset of admissible states that satisfies the three conditions is called *recovery region*
- Lyapunov function inside state constraint polytope represents recovery region
- Geometrically, the function defines an n-dimensional ellipsoid
- Important property is, for a given controller, if system state is in ellipsoid, it will stay there and converge to equilibrium point (set-point)
- So obviously use boundary of ellipsoid as switching rule

Lyapunov function

- Not unique for a given system-controller combination
- Mathematically, use linear matrix inequality method to find largest ellipsoid in polytope
- Use Lyapunov theory and LMI tools solve recovery region problem (they used a Stanford tool to find largest ellipsoid)
- You guys aren't going to remember the math behind it, so read the paper if you are curious (also it's brief and unjustified)

Additional notes

- HAC also protects plant against latent faults in HPC software that tests and evaluations fail to catch
- In certain applications like chemical-process control, where there is typically no precise plant model, might have to codify recovery region experimentally
- Stability envelope of HPC generally smaller than that of HAC, so HAC will not restrict HPC
- Examples of forward recovery using feedback
  - *Ethernet*: correcting occasional collision easier than preventing them
  - *TCP*: correcting occasional congestion better than avoiding it
  - *Democracy*: mechanism to remove undesirable leaders better than relying on infallible leaders