APOGEE RESEARCH

Analyzing Code Stability Using Control Theoretic Techniques

Jessa Lee

Mikey LiBretto

Ben Holland

Evan Fortunato

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

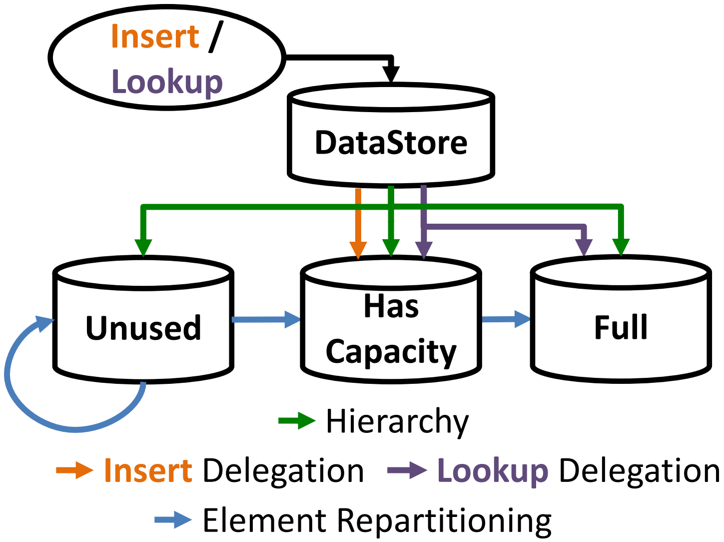
Overview

- We Will Discuss the Application of Control Theory to Software
  - Control Theory studies the behavior of dynamical systems. For example, control theory describes the conditions under which an inverted pendulum will not fall over
  - Software describes a dynamical system – can we apply control theory?
- Controller-Oriented Programming (COP) is a New Programming Language Paradigm Developed to Enable Software that is Efficient & Adaptable
  - Adds two key language constructs: Partitions and Controllers
    - Partitions capture sets of implementation options that can be treated as equivalent
    - Controllers dynamically select among these options and manage side effects and other couplings to enable systems to act like they are decoupled
  - Separates *action flow,* which specifies the essential tasks necessary to provide the required functionality, from *controller flow*, which restores necessary pre-conditions
  - Hypothesis: partitions and controllers and the resulting separation of action and controller flow may lead to ability analyze more easily
  - SymLang is the first instance of a COP Language
- We will describe an example control theory-based analysis of SymLang code

Simple Example Problem: Data Stores Utilizes Feedback Control to Ensure Sufficient Resources

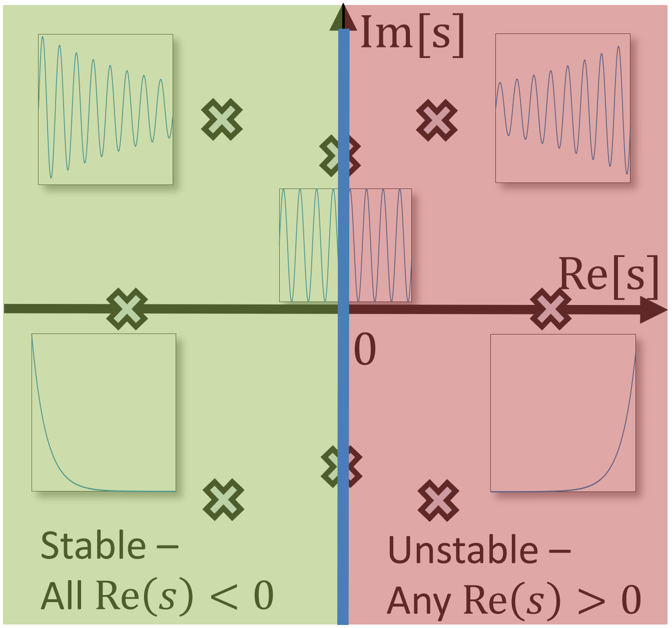
APOGEE RESEARCH

- Data Store Problem: given an unbounded stream of integer values, support lookup (true iff the value has been seen previously) in bounded time
- DataStore Implementation uses an unbounded set of atomic stores
  - Stores are organized into 3 partitions
    - Unused: new stores; uses a feedback controller to spin up additional stores as current stores are depleted
    - HasCapacity: stores with capacity; supports insert and lookup
    - Full: full stores; supports lookup
  - Insert is implemented by...
    - Inserting the value into a store in HasCapacity, which also triggers a controller to
      (a) move the store to Full, if it not longer has remaining capacity, and
      (b) take a store from Unused if HasCapacity becomes empty as a result
    - A controller also spins up new stores in Unused in anticipation of future needs



A Brief Overview of Types of Stability

- BIBO (bounded-input, bounded-output) Stability: System is bounded by a finite output for a finite input
  - Example: Ideal oscillator – when displaced, oscillates with finite amplitude around its equilibrium
  - Asymptotic stability: System returns to equilibrium when displaced
    - Example: pendulum with friction, when displaced will always trend back to its downward position
    - Condition for LTI systems: all poles have Re(s)<0
- Marginal stability: Displaced system does not explode but also does not return to equilibrium
  - Example: mass on a surface with friction – when impacted, it will travel and stop eventually but won't return to its original position
- Unstable: Displaced system explodes
  - Example: Mic and Speaker – the roar of positive feedback when a mic picks up the speaker output



Poles of the Transfer Function Indicate Stability

- Transfer functions describe Input-Output relationships of a system:

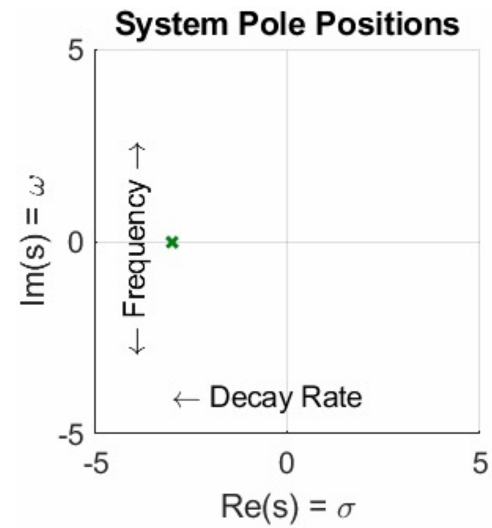
$$H(s) = \frac{output(s)}{input(s)} = \frac{N(s)}{D(s)}$$

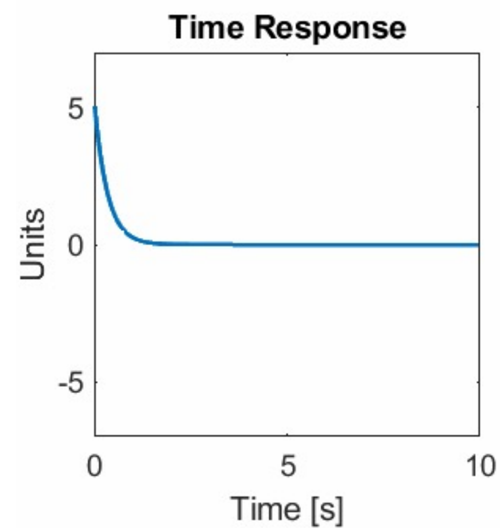
- Poles are $s \in \mathbb{C}$ s.t. $D(s) = 0$

Control Theoretic Approach to Stability: Transfer Function Analysis

APOGEE RESEARCH

- Transfer Function Analysis Provides a Simple Way to Analyze Stability of Linear Time-Invariant Systems
- Step 1: Create a block diagram capturing system dynamics
  - Block diagrams live in the Laplace domain
  - Fourier transforms decomposes a signal into frequency components (sines and cosines): $e^{i\omega t}$
  - Laplace transforms include both real and imaginary components to capture signal growth in addition to oscillations: $e^{(\sigma + i\omega)t}$
    - $\sigma > 0$: signal blows up: $for\ \sigma > 0, t \to \infty\ e^{\sigma t} \to \infty$ (unstable)
    - $\sigma < 0$: signal decays: $for\ \sigma > 0, t \to \infty\ e^{\sigma t} \to 0$ (stable)
    - $\sigma = 0$: signal oscillates forever (neither stable nor unstable)
- Step 2: Solve for the Transfer Function, $H(s) = \frac{output(s)}{input(s)}$
  - Relates the output signal to the input signal
  - Derived by reducing block diagram (well-understood in Control Theory)
- Step 3: Analyze the poles of the transfer function
  - A system is stable if and only if all poles have negative real part
  - Otherwise, the system does not converge (conceptually, the output blows up for a non-decaying input)





Control Theory is Designed to Analyze Stability… Why Not Apply to Code?

APOGEE RESEARCH

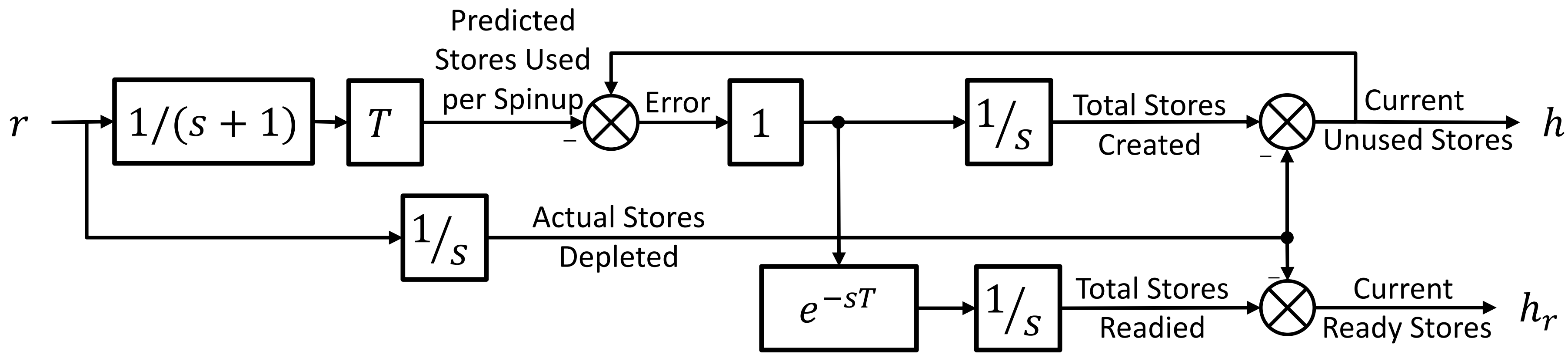
- Software Is a Dynamical System
  - Inputs are transformed into outputs
  - Software defines these transformations in code
- Failures at Cloud-Scale Often Look like Stability Issues
  - Amazon – 2021, large-scale AWS outage due to an internal migration that caused a temporary spike in network activity, which became self-perpetuating due to retry (e.g., on timeout) policies
  - Microsoft Azure – 2018, Overloaded Redis cache increased lookup latency, leading to application-level timeouts, which caused cascading failures, leading to a 17-hour downtime for multi-factor user-authentication
- Control Theory Answers Questions That Seem Relevant for Software
  - Stability: does the system have a bounded output for all sequences of bounded input?
  - Margin: does the system have sufficient resources such that future stability is guaranteed? (Is it possible for the system to run out of resources (in the future)?)
  - Note: This analysis describes the conditions under which we can guarantee that stability holds
  - Challenge is bridging the gap between control theory tools and code implementations

Controllers in (SymLang!) Code

- SymLang Code Incorporates Controllers As First-Class Language Elements, Defines a Dynamical System With Separation of Concerns
  - *Action flow* specifies the essential tasks necessary to provide the required functionality, while *Controller flow* which restores necessary pre-conditions
  - For example, in the Data Store Insert Implementation,
    Action flow specifies that the value is inserted into a store to support lookup
    Controller flow ensures that HasCapacity has a store and can support insert
- For Data Store Implementation, Want to Analyze the Stability of the Number of Unused Stores
  - Want to show that there does not exist a condition under which the number of Unused stores could become unbounded
  - Bounded input bounded output (BIBO) stability would guarantee that, for any bounded input rate, the number of unused stores is always bounded
- Question: Can We Apply Control Theoretic Techniques to the SymLang Code to Show BIBO Stability?

Derived Block Diagram From Code, Transfer Function for Stability Analysis

Step 1, Manually derived a Block Diagram for the DataStore Implementation



Step 2, derived a Transfer function describing the number of unused stores as a function of insert rate:

$$H(s) = \frac{output(s)}{input(s)} = \frac{h}{r} = \frac{T - (1 + s)}{(s + 1)^2}$$

Step 3, Evaluated the poles by solving $D(s) = (s + 1)^2 = 0$

$$s = -1$$

Because s has negative real part, this implies BIBO stability

$$Re(s) = -1$$

∴ for any bounded insert rate, the number of unused stores will remain bounded

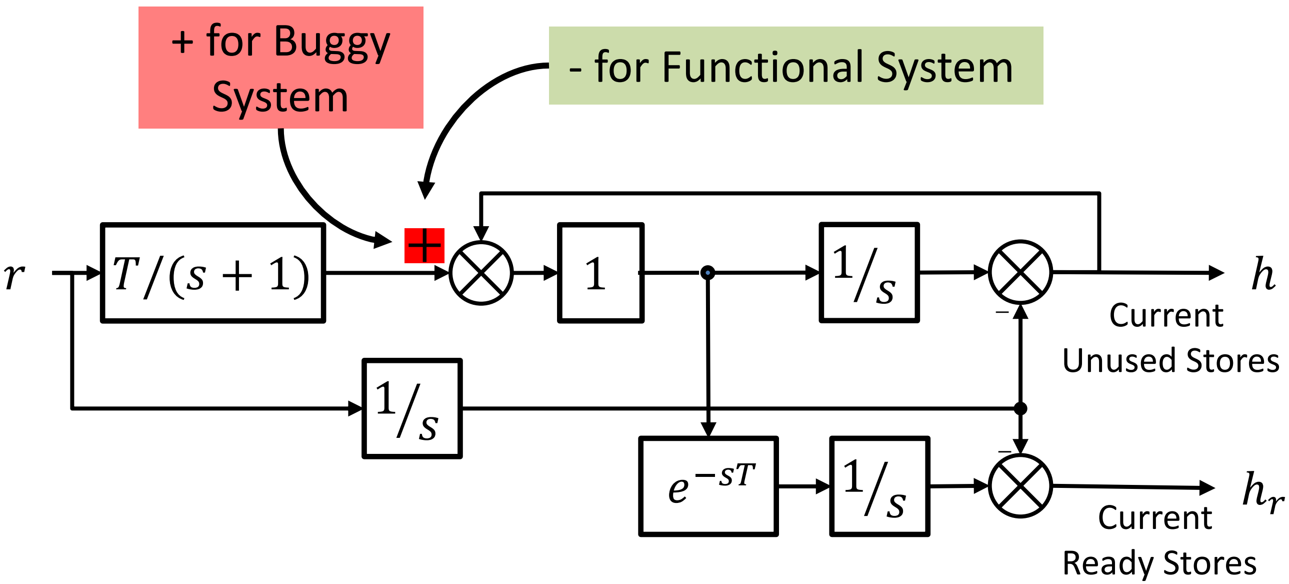
Analysis Correctly Identifies Unstable Implementations: Positive Feedback Bug

APOGEE RESEARCH

- Analyzed Two Real Bugs Introduced by Junior Devs

```
Bug #1
// creates positive feedback
val error = (setpoint + size())
```

```
Functional Implementation
internal void runPID() = Do {
  action() = {
    val error = (setpoint - size())
    var desiredUpdate = PID.execute(error);
    addOrRemoveStores(desiredUpdate);
}}
```



Transfer Function

$$\frac{h}{r} = \frac{T-(1+s)}{(s+1)(s-1)}$$

Pole at $s = +1$, so System is UNSTABLE

Transfer function analysis shows positive feedback creates instability

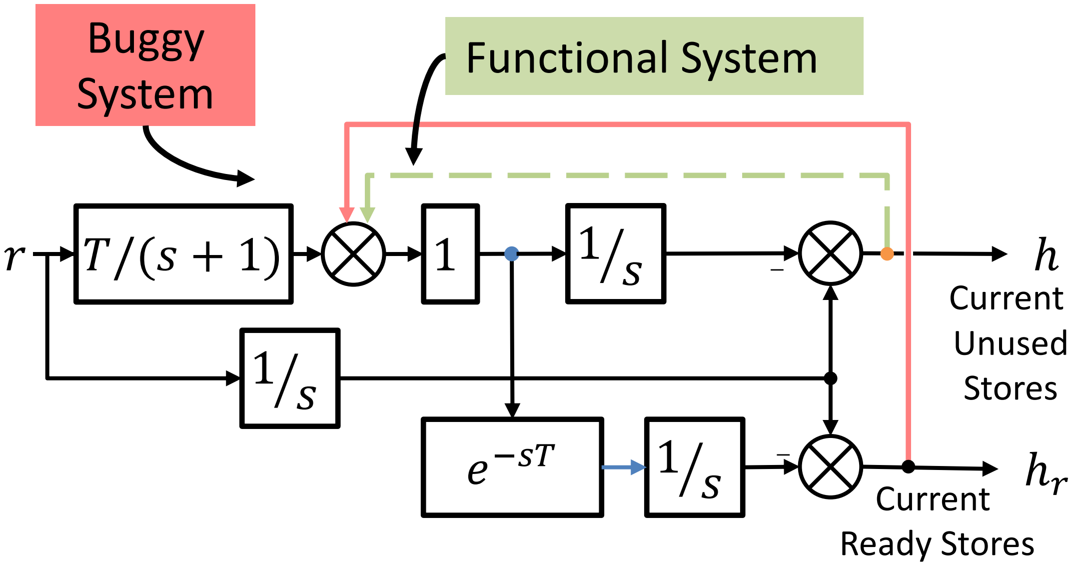
Analysis Correctly Identifies Unstable Implementations: Latency Bug

Bug #2

```
// introduces latency by only
// counting Ready stores
val error = (setpoint – Ready.size());
```

Functional Implementation

```
internal void runPID() = Do {
  action() = {
    val error = (setpoint - size())
    var desiredUpdate = PID.execute(error);
    addOrRemoveStores(desiredUpdate);
}}
```



Transfer Function

$$\frac{h}{r} = \frac{sT + (s+1)(1 - s - e^{-sT})}{s(s + e^{-sT})(s+1)}$$

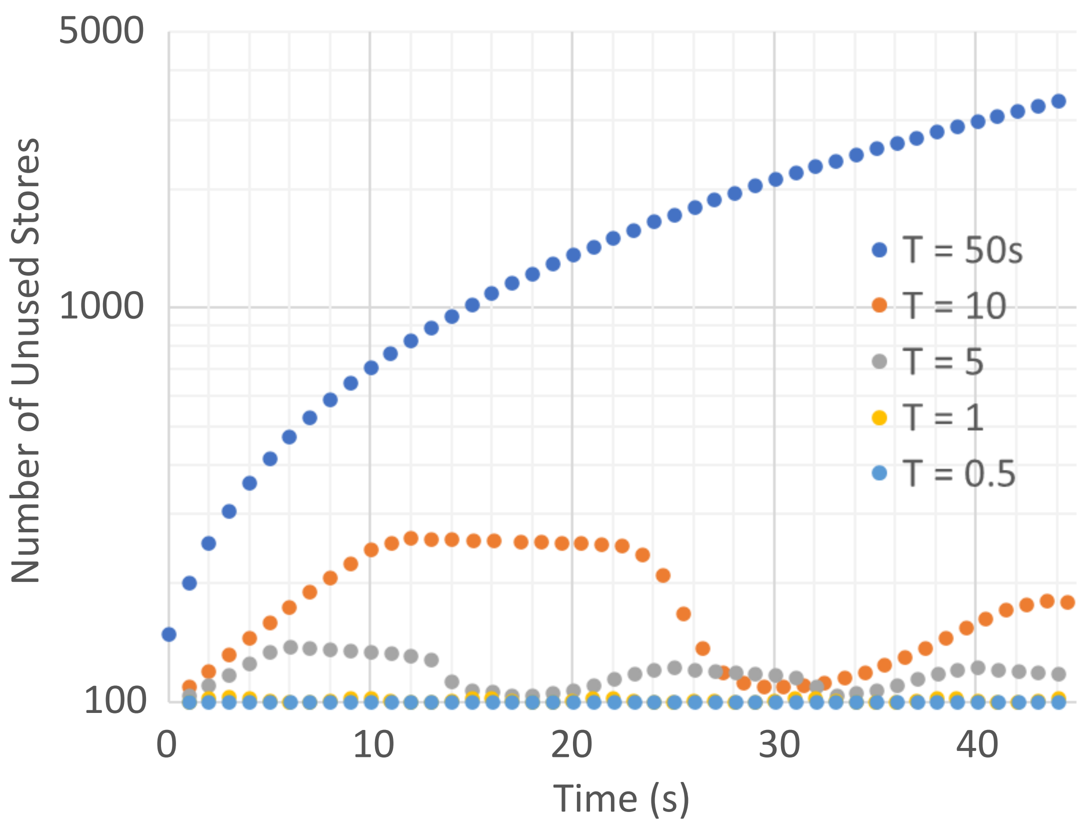
Poles at $s = 0, -1$ and s s.t. $s + e^{-sT} = 0$

For $T \geq \frac{\pi}{2}$, $s + e^{-sT} = 0$ has roots with positive real part, so System is UNSTABLE

Transfer function analysis shows introduced latency creates instability

Control Theory Identifies Regions of Stability and Instability

- With Bug #2, DataStore has regions of stability, instability
  - When T (spin-up time) is small, DataStore appears stable
  - For large T, number of stores grows without bound
  - T is the time to spin up a new store – an external parameter. If cloud outage causes an increase in latency, do not want the system diverge unrecoverably!



- Control Theory Reasons over the Range of Possible Spin up Delays
  - Static analysis that identifies potential instabilities due to non-syntactic errors
  - Provides stronger confidence than running a small sample of points in the config space

Moving Towards Automated Analysis

- Automated Stability Analysis Combines Techniques from Code Analysis and Control Theory
  - Tool uses data flow, control flow, and Laplace transforms of controller functionalities to derive the block diagram
  - Analysis of stability from a block diagram is well-understood in control theory
- Support Controller Analysis for a Limited Subset of Language
  - Automated controller analysis not possible in general, e.g., code must be analyzable
  - Defined a restricted set of primitive operations such that anything written in this subset of the language can be analyzed. Next step: formalize as a restricted DSL
  - SymLang also provides Control Theory Libraries for standard functionality, e.g., PID controllers, that include Laplace Transforms to enable analysis
- Implemented Working Prototype of Automated Stability Analysis
  - Analyzes example DataStore implementation, and we believe the prototype will extend to other relevant cases
  - Happy to provide both the SymLang and analysis code to those interested (conditioned on government approval for release)

Conclusion

- Demonstrated Analysis of Stability of SymLang Code
  - Derived transfer function from SymLang code
  - Pole analysis correctly identified stability and instability of implementations
  - Automated analysis of SymLang implementation for example Data Store problem
- Practically, Stability Bugs are Not Easy to Catch with Current Tools
  - Bugs are semantic, not syntactic – code will compile because syntax validation, type-checking, and other common code analysis techniques do not reason about stability
  - Require reasoning over a very large (possibly infinite) state space
- Early Work – Lots More to Do!
  - Extend and improve automated analysis tool
  - Margin analysis: in addition to stability, want to know if sufficient stores available
  - Approaches for scalability: can we use compositional approaches to achieve scalability, e.g., by characterizing the gain and phase lag of each module

Questions?