Final Project

Concurrency Models Across Languages

Overview

Each of the languages we have studied takes a fundamentally different position on how concurrent computation should be structured and how the risks of concurrency should be managed. Java's model is built around shared mutable state protected by locks. Elixir's model is built around isolated processes that communicate only by passing messages. Go's model is built around channels and the CSP discipline of communicating sequential processes.

These are not just different APIs. They reflect different theories about where the danger in concurrent programming lives, what the programmer should be forced to think about explicitly, and what the runtime or type system should handle automatically.

In this project you will analyze what happens when you try to express one language's concurrency model using the facilities of a different language. The central question for each case is: what maps naturally, what requires awkward simulation, and what cannot be represented at all?

The goal is not to write large amounts of code. Small, precise code fragments are appropriate where they illustrate a specific point. The goal is analytical writing that demonstrates you understand the models at a level deeper than their syntax.

Policy on AI and your own work

*This project is individual work and your own work. You are bound by the honor code at UNC to do the work under the policies outlined here. You **may use** your notes and class materials. You **may use** AI tools and online resources as quick efficient syntax and semantics checkers, but you **may not** "look up" solutions to these questions, or simply use AI tools to produce solutions for you.*

The Three Models

Shared memory (Java)

Threads share a single address space. Coordination is achieved by controlling access to shared data through locks, monitors, or higher-level constructs such as those in java.util.concurrent. The programmer is responsible for identifying shared state and ensuring that all accesses are properly synchronized. Correctness depends on programmer discipline; the language and runtime do relatively little to prevent unsynchronized access.

Actor model (Elixir / Erlang)

Processes are isolated; no memory is shared between them. All communication is through asynchronous message passing to a process's mailbox. Mailboxes are heterogeneous and unbounded. The receive construct uses pattern matching to select the next message to process and can skip over non-matching messages, leaving them in the mailbox for later. The OTP

supervision model provides structured fault tolerance: a supervisor process monitors worker processes and restarts them on failure, implementing the “let it crash” philosophy.

CSP / channels (Go)

Goroutines are lightweight concurrent units that communicate through typed channels. An unbuffered channel provides rendezvous synchronization: the sender blocks until a receiver is ready, and vice versa. The select statement allows a goroutine to wait on whichever of multiple channel operations becomes ready first, with random selection among simultaneously ready cases. Go permits shared memory and provides sync.Mutex, but the idiomatic Go style strongly favors communication over sharing.

Part 1: The Model Matrix

Your task is to (for example) design how Go channels can be implemented using Java shared memory threads. And then do the same for the other meaningful combinations (since Java is Java is not meaningful… Go in Go, Actors in Elixir).

The table below names the six non-trivial combinations. For each cell where the model is not native to the language, write an analysis of the following three questions. Each cell’s analysis should be roughly 250–400 words plus any illustrative code fragments.

Model being expressed → Language used ↓	**Actor model (Erlang-style)**	**CSP / channels (Go-style)**	**Shared memory (Java-style)**
Java	Native threads + locks (baseline)	Simulate with BlockingQueue; select has no direct equivalent	Native synchronized / java.util.concurrent (baseline)
Elixir	Native GenServer / spawn (baseline)	Simulate synchrony with call/reply; no true rendezvous	Impossible on the BEAM; simulate with Agent or ETS
Go	Simulate with goroutine + chan interface{}; selective receive lost	Native goroutines + channels (baseline)	Possible with sync.Mutex but idiomatically discouraged

The three shaded diagonal cells (Java/shared-memory, Elixir/actor, Go/CSP) are the native implementations. You are not required to analyze these beyond using them as your baseline reference. Your written analysis covers the six off-diagonal cells.

Also note that for Go, we want to analyze making *Go channels*. Go has the same shared memory and mutex facilities that Java has, so we want to know how to do Go channels in Java, and Go channels in Actors/Elixir.

For each non-native cell, address:

1. What maps naturally

Which features of the host language make this model easy to approximate? What do you get for free? Be specific: name the construct and explain why it corresponds to a concept in the target model.

2. What requires simulation or workaround

Which features of the target model have no direct equivalent in the host language, forcing you to build something by hand? Describe the simulation, explain what it costs (in complexity, performance, or correctness risk), and explain what is lost in translation. A simulation that works but requires manual discipline where the native model provides automatic enforcement is a significant observation.

3. What cannot be represented well or at all

Which features of the target model are architecturally foreign to the host language — not just missing a convenient API, but in fundamental tension with the host language's design? Examples might include features that the runtime prohibits, features that require giving up the host language's type safety, or features whose absence changes the correctness guarantees available to the programmer.

Feel free to express your ideas using code fragments and examples if you wish, but we are not looking for any full implementation examples or extended code.

Possible Dimensions of Analysis

As you discuss your choices of how to represent one model in another language, consider indicating some of these issues and considerations:

1. *Communication Mechanism*

How do concurrent entities exchange information? What guarantees exist (ordering, typing, isolation)?

2. *Message / Event Selection*

How does a system decide what to handle next? Is there selective receive, select, or only blocking operations?

3*. Coordination and Synchronization*

How do entities coordinate behavior? What mechanisms enforce ordering or mutual exclusion?

4. *Blocking and Progress Semantics*

When does computation block? What guarantees exist about progress, deadlock, or fairness?

5. *State Management*

Where does mutable state live? What guarantees or risks follow from this design?

6. *Failure and Recovery*

How are failures handled? Is there supervision or restart? How must it be simulated?

Guidance and examples

The following are examples of the kind of specific, mechanistic observations being looked for. These are starting points, not a complete list. Strong papers will go beyond these.

- Actor model in Go: Erlang mailboxes are heterogeneous and support selective receive — the receive expression pattern-matches and skips non-matching messages, leaving them in place. Go channels are typed and strictly FIFO. Simulating selective receive requires building your own queue with a scan loop, which is O(n) in the mailbox size and loses the elegance of pattern-matched dispatch entirely. Using chan interface{} recovers heterogeneity but at the cost of compile-time type safety.
- CSP in Java: java.util.concurrent.BlockingQueue is a reasonable approximation of a buffered channel. The harder problem is Go's select statement, which synchronizes on whichever of N channel operations is ready first, with a defined semantics for simultaneous readiness. Java has no direct equivalent. Simulating it requires either polling (introducing latency and CPU waste) or a shared multiplexing queue (which loses directionality).
- Shared memory in Elixir: The BEAM does not permit shared mutable state between processes. This is not a missing library; it is a runtime constraint. An Agent wraps state in a process and serializes access through message passing, which is not shared memory — it is the actor model applied to a stateful value. The simulation works but eliminates the defining characteristic of the shared-memory model: direct, low-latency access to a common address space.
- Actor model in Java: The let-it-crash philosophy has no natural home in Java. Java's instinct is to catch exceptions locally and recover at the site of failure. Erlang's instinct is to allow the process to die and rely on a supervisor to restart it with clean state. Building a supervisor in Java requires explicit thread monitoring, restart logic, and state initialization — all of which OTP provides as a framework. The simulation is possible but makes the programmer responsible for what OTP guarantees automatically.

Part 2: Best and Worst Judgments

Based on your analysis of the matrix, make and defend two judgments.

2a. Best overall fit

State which of the 6 concurrency model representations you judge to be the best: the most easily and naturally represented, the most fully matching in semantics of the original model you are representing… whatever you view as relevant. State the criteria by which you are judging.

2b. Worst overall fit

State which cell/representation you judge to be the "worst fit" by your chosen criterion (or a different criterion, if you prefer — state it explicitly). Which one was least natural, or missed the most aspects of the original semantics, or required more infrastructure to construct. Apply the

same requirements: state the criterion, cite evidence, acknowledge and rebut the counterargument.

Part 3: Synthesis

Write a short synthesis section (approximately 300–500 words) that addresses the following question:

> *What does the difficulty of expressing each model in foreign languages reveal about the nature of the models themselves? Are some models more "portable" than others, and if so, what does that tell you?*

You might find it useful to think about whether portability correlates with how much a model relies on runtime support versus language-level constructs, or with how much the model depends on a particular theory of where errors come from and how they should be handled.

This section should draw on your matrix analysis rather than introduce new claims from scratch. Think of it as the conclusion that the matrix evidence supports.

Submit as a single PDF or Word document with any code in the document. I wont be compiling or running code to grade this. Code should be readable: use a monospace font, reasonable line lengths, and enough whitespace that structure is visible.

Grading

Section	What is being assessed	Points
Part 1: The model matrix (6 cells)	Depth and accuracy of analysis for each non-native pairing; use of specific mechanisms as evidence	48 (8 per cell)
Part 2: Best / worst judgments	Clarity of stated criterion; quality of argument; engagement with counterarguments	20
Part 3: Synthesis	Insight and coherence; connection back to matrix evidence; quality of writing	20
Overall clarity and rigor	Precision of language; avoidance of vague claims; appropriate use of examples	12
Total		**100**

The matrix is worth the most because it is where the core analytical work lives. Within the matrix, depth matters more than breadth: a single cell with a precise, well-evidenced argument about a fundamental mismatch is worth more than three cells with vague generalizations.

For the best/worst section, the quality of the argument matters more than the choice of position. Two papers can pick opposite positions and both score full marks if both engage seriously with the evidence and the counterarguments.

Advice

- Vague claims lose points. “Elixir is good at fault tolerance” is not an observation; “Elixir’s supervisor trees restart failed processes with clean state, and there is no equivalent abstraction available in Java without building it by hand” is an observation.
- Distinguish between shallow and deep mismatches. A missing convenience function is a shallow mismatch. A runtime constraint that makes the concept architecturally foreign (like shared mutable state on the BEAM) is a deep mismatch. The deep mismatches are the most interesting.
- Use your baseline implementations as evidence, not decoration. If you claim that selective receive is awkward in Go, show the workaround in your pipeline code and explain what it cost.
- For the best/worst section, state your criterion before your conclusion, not after. “X is best; my criterion is Y” reads as post-hoc rationalization. “My criterion is Y; by that criterion X is best because…” is an argument.
- The synthesis section is not a summary. Do not restate observations from the matrix. Use them as evidence for a new, higher-level claim.