**Distributed Concurrency Geheugensteuntje**

* **Stateless Model checking**: Validate system correctness by systematically exploring all possible executions. Struggles with state space explosion. Uses state space reduction techniques. Different from my method in that I will not exhaustively search.
* **Fault/Crash Injection, Network partition:** Many systems focus on inserting faults, crashes network partitions as well as message reordering. Beyond the scope of my research.
* **Local Concurrency vs Distributed Concurrency:** Some papers focus on multithreaded programs instead of distributed programs.

**Concurrency Testing Papers**

* Learning-based Testing: Pick next event based on QLearning. Goal is to get high coverage of program under test. Different abstractions H(s) of program state as feedback to the algorithm, based on hashing individual nodes’ program state. Highest abstraction is just the ‘inbox’ of enabled events per machine.
  + Limitations: Highly reliant on good abstraction H. Parameter tuning required.
* MoDist: Stateless Model checking. Replaying trace to force system into state.
  + Limitations: Requires significant state information (white-box). Failure injection
* SAMC: Multiple failure injection. Model checker. Using white-box information for state reduction.
  + Limitations: TODO
* FlyMC: White-box Model checker with state-space bounding. Communication and state symmetry: Explore schedule for a single role type, irrespective of node id. Event independence: considers single process events that update disjoint variables as independent. Parallel flips: Flips pairs of events per node at the same time instead of one node per run.
  + Limitations: Not search-based. White box.
* DCatch: DC bug detector using HB relations. Looks for concurrent memory access triggered by distributed events. Creates HB graph and compares this to memory access for identifying bug candidates, then further prunes this.
  + Limitations: Memory-access state required. Lots of specific HB-relation for different systems. Will not include.
* dBug: Both shared-memory and distributed systems reducing state-space through DPOR
  + Limitations: Requires corrections check, and interface to system non-determinism. model-checking. Run-time overhead from VM snapshots and steady state detection. dBug perturbs SUT.
* Morpheus: Randomized concurrency testing. Using POS with conflict analysis. Keep table of operations to (non-)conflict. Schedule non-conflicting operations immediately.
  + Limitations: Conflict analysis requires low-level access to operations and scheduling. Only for Erlang code. Relies on custom-built test cases and high-level invariants.
* DeMeter: Dynamic Interface Reduction (DIR). State-space reduction for model-checking by 1. Checking components separately and 2. Discover interface behavior dynamically.
  + Limitations: Implementation-level model-checker (white-box).
* TraceAware: Randomized concurrency testing. Using POS and PCT. Uses dynamic analysis to approximate set of racy events by running the SUT several times up to n events. Static analysis to identify commutativity of co-enabled events in the same node. i.e., racy events that are in fact not racy.
  + Uses static analysis to further reduce the set of racy events.
* CommClosure: Reduce asynchronous semantics to synchronous through communication closure. Requires heavy synchronization and instrumentation of code. Randomized testing, sampling from the reduced number of executions.
  + Limitations: Requires heavy instrumentation to ensure synchronous execution. Not applicable to systems that violate the communication closure. l

**Ideas:**

Use priorities in GA. Assign priorities to events and execute events based on that priority.

* Challenge: How long do we wait after the execution of an event for other events to come in.
* Solution: Both a timer, maximum time a message can be in the inbox, and a maximum number of enabled events at one time. Above any limit means executing more events. Can even be a dynamic parameter.

Reduce non-determinism in initial state at each test case by assigning node numbers based on the order at which they start a new consensus round. All nodes have the same role, and their order shouldn’t matter. This should also produce trace graphs with more similarity.

Reduce non-determinism even more by having the testing algorithm call StartRound through a custom message. This completely syncs the rounds of all nodes. The original StartRound call would inform the tester of the readiness of the nodes for the new round. See EndConsensus -> BeginConsensus in NetworkOps -> StartRound in RCLConsensus -> startRound in Consensus. Find suitable place to notify tester and wait for message from tester to call startRound.

Need to control timerEntry for each node separately to have any chance of finding a bug. Executing an event has no effect until timerEntry is called in a node. Only then is progress made in the consensus phase. Is there an easy way to control this? Use a client command or peer message to call NetworkOPs::processHeartbeatTimer and don’t call setHeartbeatTimer. What is the policy for heartbeating different nodes? Need to think about liveness/synchronization issues stemming from decoupling the heartbeat timers from the clock. Don’t allow two subsequent hearbeats to one node without heartbeating all other nodes? Treat heartbeats as an event with extra constraints.

Can I use racy events in my algorithm? We want to reorder racy events. Record racy events during test case and their order. All events in ripple are racy…

Objectives of the search.

* Time taken to get transactions into ledger.
* Validate transactions that are in ledgers
* Create test case that tries to double spend tokens.
* Try out new traces with each run, how?
  + Use trace graph distance to get a diverse set of traces and explore further
* How can I encode ordering of racy events?
  + Record racy consensus events and their orders of previous executions, by
    - Storing tuples/lists with events that race. Idea is that receiver does not matter due to symmetry of roles. How to reduce this sender redundancy? Count identical and distinct senders? Example with 3 nodes n = {0, 1, 2}. Say three racy events: e1=(0,2,validation), e2=(0,2,validation), e3=(1,2,validation). Store tuple <(0, validation), (0, validation), (1, validation)>. This also counts if e1=(2,0,validation), e2=(2,0,validation), e3=(1,0,validation) (would be stored similarly to first racy event set).
    - Ask Burcu if she thinks detecting racy events is a good idea.
  + Pre-run where racy events are dynamically collected. Then apply prioritization and/or delays to these events only.
  + Do not want to over specialize racy events to ripple code, so just use normal racy event condition. How to detect racy events? For each event, find all events co-enabled with this event. Then from those events check whether the event is dependent. Use trace graph for this?

Potential ripple bug. Don’t check whether the sequence number of a peer’s proposal is higher than the previous peer’s proposal.

Changes required for inbox scheduler.

* Scheduler trait
  + Start(ga\_receiver<DelayMapPhenotype>) -> DelayMapPhenotype should be trait?
  + execute\_event() -> Default implementation
  + Listen\_to\_peers() -> extrapolate to only receiving messages and put in inbox. Delay will immediately ‘schedule’ messages that arrive in inbox. Work with condvar and mutex to allow different function to act on inbox contents.
  + Is\_consensus\_rmo() -> default implementation
  + Listen\_to\_ga() -> leave for specific implementing structs
  + Update\_current\_round() -> default implementation
  + Update\_latest\_validated\_ledger() -> default implementation
  + Harness\_controller() -> default implementation, refactor test\_harness after to not continually parse a file…
* Inbox
  + What data structure to use
    - Arc Mutex since it’s shared between listen\_to\_peers schedule\_controller.
    - Sorted list insertion and head removal.