Model Checking in RAMCloud

Diego Ongaro
2012-02-10

Why should we care?

- RAMCloud contains many protocols that are hard to reason about
  - Log entries: tombstones, cleaning, recovery
  - Distributed protocols: tablet migration
  - Finding log segments during recovery
- Model checking is like unit testing but for protocols
  - Might catch some bugs
  - Will definitely make you think more
- Helps to formalize complex invariants

Overview

- Define model checking
- Toy example: traffic light
- RAMCloud example: finding log segments during recovery
- Discussion

What is model checking?

- You define:
  - A set of variables (state)
  - Starting values
  - Rules for transitioning between states
  - Invariants
- The model checker (e.g., Murphi) will:
  - Brute force to find all reachable states
  - Ensure all invariants always hold
- Similar to NFAs
- Main challenge: state explosion

A toy example: traffic lights

- A protocol involving traffic lights and drivers exists to avoid collisions
- Let's model an intersection with two one-way streets and two traffic lights
- Assume the lights and drivers obey the protocol
- Show that no two cars will occupy the intersection at once

State definition

- Two lights
  - A light is *red* or *green*
- Set of cars
  - A car can be *elsewhere*, *waiting* at a light, or *crossing* at a light
- An array indexed by a scalarset enables symmetry reduction

```
const
  NUM_LIGHTS: 2;
  MAX_CARS: 5;

type
  LightId: scalarset(NUM_LIGHTS);
  CarId: scalarset(MAX_CARS);
  Car: record
    state: enum { ELSEWHERE, WAITING,
                  CROSSING };
    light: LightId;
  end;

var
  lights : array [ LightId ] of enum { RED, GREEN };
  cars : array [ CarId ] of Car;
```

Traffic light rules

- Start state: lights are *red*, cars are *elsewhere*
- Light transition rules
  - If some light is *red* and the other is *red*, change it to *green*
  - If some light is *green* and there are no cars in the intersection, change it to *red*
- Car transition rules
  - If some car is *elsewhere*, change it to *waiting* at some light
  - If some car is *waiting* and the light is *green*, change it to *crossing*
  - If some car is *crossing*, change it to *elsewhere*

- Can you spot the bug? Murphi did (next slide)

Murphi's diagnosis

- Invariant "No collisions" failed.
- Startstate Startstate 0 fired.
  - lights[LightId_1]:RED
  - lights[LightId_2]:RED
  - cars[CarId_1].state:ELSEWHERE
  - cars[CarId_1].light:Undefined
  - cars[CarId_2].state:ELSEWHERE
  - cars[CarId_2].light:Undefined
- Rule car arrives at some light, l:LightId_1, c:CarId_1 fired.
  - cars[CarId_1].state:WAITING
  - cars[CarId_1].light:LightId_1
- Rule car arrives at some light, l:LightId_1, c:CarId_2 fired.
  - cars[CarId_2].state:WAITING
  - cars[CarId_2].light:LightId_1
- Rule light changes to green, l:LightId_1 fired.
  - lights[LightId_1]:GREEN
- Rule car begins crossing, c:CarId_1 fired.
  - cars[CarId_1].state:CROSSING
- Rule car begins crossing, c:CarId_2 fired.
  - The last state of the trace (in full) is:
  - lights[LightId_1]:GREEN
  - lights[LightId_2]:RED
  - cars[CarId_1].state:CROSSING
  - cars[CarId_1].light:LightId_1
  - cars[CarId_2].state:CROSSING
  - cars[CarId_2].light:LightId_1

Workflow

- Identify the protocol you're targeting
- Think through simplifying assumptions
- Distill the state variables to the very essentials
- Write your invariants and rules
- Plan to spend most of your time refactoring
- Run the model checker
- Extract your invariants and algorithms from code into plain English

Finding log segments during recovery: problem statement

- Each RAMCloud master replicates segments of its log across the entire cluster
- When a master crashes, the coordinator must find at least one copy of every segment replica that made up the master's log (*recoverability*)
- It must also ignore segment replicas that are no longer part of the master's log (freed by the cleaner or stale due to backup partitioning)

Ned to find the head segment

- Store a *log digest* in each segment, listing every segment that is currently in the log
- Only the log digest in the head segment is valid
- So if we can find a replica of the master's head segment, we can find the exact set of segments that make up the log

How do we find the head segment?

- Open before close protocol on segments
  - Durably open new head segment
  - Durably close old head segment
  - Begin writing to new head segment
- During recovery, if there are two open segments, it doesn't matter which one is used.

Problem #1: Initialization

Problem #2: Backup failures

Simplifying assumptions

- Each master's log is independent
- We don't need to model backups; each replica can be stored independently
- Assume RPCs are asynchronous but atomic
- We don't need 8 MB segments
- We shouldn't model the cleaner right away
  - So we don't need to model log digests
- Model failures to maintain recoverability

Invariant

- Given an adversarial set of replicas to use in recovery, the log recovered contains exactly:
  - the objects durably written by the master
  - optionally, any objects the master was in the process of writing
- Adversarial set:
  - For each segment, use its shortest replica
  - Stop at the earliest replica that looks like a head segment

State

- The log:
  - An array of segments, indexed by segment ID
  - A *ready* boolean, indicating whether the master may have tablets assigned to it
  - *minValidOpenSegment:* During recovery, all open segment replicas with segment IDs below this number are ignored.
- Each segment
  - May be invalid, open, or closed
  - Has a length
  - Has a set of replicas
- Each replica:
  - May be invalid, open, or closed
  - Has a length
  - May be partitioned (unavailable for replication)

Rules

- Start state: empty log, no replicas
- Writing to the log
  - If the log is ready, the head segment has room and is durable, and the previous segment has been durably closed: *increase the length of the head segment.*
  - If the current segment is full: *mark it closed and mark the next segment open.*
- Replication
  - If a segment needs replication, one of its available replicas is out of date, and the segment is open or the next segment is durably open: *update the replica.*
  - If a segment needs replication, needs a new replica to achieve this, and the segment is open or the next segment is durably open: *create a new, up-to-date replica.*
- Partitioning a replica (backup failures)
  - If a replica is open and not partitioned: *partition it. If the segment is open, mark the segment closed, mark the next segment open, and set the backup recovery flag.*
  - If the backup recovery flag is set, the head segment is durably open, and all previous segments are durably closed: *set the minValidOpenSegmentId to the head.*

Why won't this catch every bug?

- Your simplifying assumptions might hide bugs
- Your model might be buggy
  - Tip: run with -pr flag for a count of how many times each rule fires
- Your invariants might not be strong enough
- Your model might be too small
- The model checker might be buggy
  - Murphi is basically unmaintained

Early lessons

- Focus on what you're trying to show
- Keep variables small
  - 2^(number of bits of state) is an upper bound on the number of states
- Take advantage of symmetry reduction
- Models tend to terminate quickly or never
- Murphi gotchas:
  - Syntactically picky
  - Need to loop around dead ends to the start state

Summary & next steps

- Model checking is like unit testing but for protocols
- Helps to formalize complex invariants
- Will scale to non-trivial examples
- Next steps:
  - Converge this model and ReplicaManager
  - Model other parts of RAMCloud
    - Log entries: tombstones, cleaning, recovery
    - Distributed protocols: tablet migration
  - Look into other model checkers