Testing stateful, concurrent, and async systems using test.check

Eric Normand

PurelyFunctional.tv
Outline

• Example-based testing is inadequate

• Generating test data

• Generating sequential tests

• Adding parallelism
A stateful example

• A key-value database

• Operations:

  (db/create)
  
  (db/clear!  db)
  (db/store!  db k v)
  (db/delete! db k)
  (db/fetch   db k)
  (db/size    db)
Let’s test it!
DB should contain a key/value after storing

(deftest store-contains
DB should contain a key/value after storing

(deftest store-contains
  (let [db (db/create)
        k "a"
        v "b"]
    ))
DB should contain a key/value after storing

(deftest store-contains
  (let [db (db/create)
        k "a"
        v "b"]
    (db/store! db k v)
  ))
DB should contain a key/value after storing

(deftest store-contains
  (let [db (db/create)
        k "a"
        v "b"]
    (db/store! db k v)
    (db/fetch db k) ))
DB should contain a key/value after storing

(deftest store-contains
  (let [db (db/create)
         k "a"
         v "b"]
    (db/store! db k v)
    (is (= v (db/fetch db k))))
store! should overwrite old values

(deftest store-overwrite )
store! should overwrite old values

(deftest store-overwrite
  (let [db (db/create)
        k "a"
        v1 "b"
        v2 "c"]
))
store! should overwrite old values

(deftest store-overwrite
  (let [db (db/create)
        k "a"
        v1 "b"
        v2 "c"]
    (db/store! db k v1)
    (db/store! db k v2)
  )
store! should overwrite old values

(deftest store-overwrite
 (let [db (db/create)
   k "a"
   v1 "b"
   v2 "c"]
  (db/store! db k v1)
  (db/store! db k v2)
  (db/fetch db k) ))
store! should overwrite old values

(deftest store-overwrite
  (let [db (db/create)
        k "a"
        v1 "b"
        v2 "c"
        (db/store! db k v1)
        (db/store! db k v2)
        (is (= v2 (db/fetch db k)))))


DB should be empty after clearing

(deftest clear-empty
DB should be empty after clearing

(deftest clear-empty
  (let [db (db/create)
        k "a"
        v "b"]
    ))
DB should be empty after clearing

(deftest clear-empty
  (let [db (db/create)
        k "a"
        v "b"]
    (db/store! db k v)
    (db/clear! db)
))
DB should be empty after clearing

(deftest clear-empty
  (let [db (db/create)
       k "a"
       v "b"]
    (db/store! db k v)
    (db/clear! db)
    (db/size db) ))
DB should be empty after clearing

(deftest clear-empty
  (let [db (db/create)
        k "a"
        v "b"]
    (db/store! db k v)
    (db/clear! db)
    (is (zero? (db/size db))))
I don’t want you to feel bad, but . . .
you should feel bad about these tests.
Not guilty, but scared.
How big is our system?

- How many strings are there?
- How many unicode characters are there?
- How many key-value pairs are there?
- How many operations are there?
- How many pairs of operations are there?
- How many triples of operations are there?
The *description* of the database is small.
Let’s set up some generators

(def gen-key gen/string)
(def gen-value gen/string)

> (gen/sample gen-key 20)

("" "Ï" "Û" "Ù" "þ7Ä" "Î" "§" "þ8zÈäè" "" "Ô"
"¢öU÷¸" "W\b^è÷¬D\" "\ngz|μβÒW." "ñø"
"ô>,βiwA,\r!" ";ÊÑ²ãÔ9" "þèIà0TzJÜ\bi"
"ó¥è¬#Àö\"ÆjF#" "u=?" "´ö")
Some easy properties...
DB should contain a key/value after storing

(defspec store-contains 100 )
DB should contain a key/value after storing

(defspec store-contains 100
  (prop/for-all [k gen-key
    v gen-value]

  ))
DB should contain a key/value after storing

(defspec store-contains 100
  (prop/for-all [k gen-key
    v gen-value]
    (let [db (db/create)]
      )))
DB should contain a key/value after storing

```clojure
(defspec store-contains 100
  (prop/for-all [k gen-key
                 v gen-value]
    (let [db (db/create)]
      (db/store! db k v)
      (db/store! db k v))))
```
DB should contain a key/value after storing

(defspec store-contains 100
  (prop/for-all [k gen-key
    v gen-value]
    (let [db (db/create)]
      (db/store! db k v)
      (= v (db/fetch db k))))))
store! should overwrite old values

(defspec store-overwrite 100)
store! should overwrite old values

(defspec store-overwrite 100
  (prop/for-all [k gen-key
                v1 gen-value
                v2 gen-value]
                ))
store! should overwrite old values

(defspec store-overwrite 100
  (prop/for-all [k  gen-key
                v1 gen-value
                v2 gen-value]
    (let [db (db/create)]
      ))
store! should overwrite old values

(defspec store-overwrite 100
  (prop/for-all [k gen-key
    v1 gen-value
    v2 gen-value]
    (let [db (db/create)]
      (db/store! db k v1)
      (db/store! db k v2)
      )))
store! should overwrite old values

(defspec store-overwrite 100
  (prop/for-all [k  gen-key
                v1 gen-value
                v2 gen-value]
    (let [db (db/create)]
      (db/store! db k v1)
      (db/store! db k v2)
      ))
store! should overwrite old values

(defspec store-overwrite 100
  (prop/for-all [k gen-key
                v1 gen-value
                v2 gen-value]
    (let [db (db/create)]
      (db/store! db k v1)
      (db/store! db k v2)
      (= v2 (db/fetch db k))))
DB should be empty after clearing

(defspec clear-empty 100
  (prop/for-all [k gen-key
                v gen-value]
    (let [db (db/create)]
      (db/store! db k v)
      (db/clear! db)
      (zero? (db/size db))))
{:result false,
  :test-var "store-contains",
  :failing-size 28,
  :num-tests 29,
  :fail ["æ"]qÜ"İ¹±WîρZ¢Ẹµgä>Å¨" "ôZãœí®"],
  :shrunken {:total-nodes-visited 139,
    :depth 33,
    :result false,
    :smallest ["æ" " "]}
  :seed 1489522410083}
Can we describe the behavior in one go?
1. Build a simple, pure model

- A key-value database is like a hash map.
2. Reify the operations and make generators

(def gen-clear (gen/return [:clear!]))
(def gen-size   (gen/return [:size]))
(def gen-store  (gen/tuple (gen/return :store!) gen-key gen-value))
(def gen-delete (gen/tuple (gen/return :delete!) gen-key))
(def gen-fetch  (gen/tuple (gen/return :fetch) gen-key))
(def gen-ops (gen/vector
               (gen/one-of [gen-clear gen-store gen-delete gen-fetch gen-size]])))
> (gen/sample gen-ops)

([]
  [[[:clear!]]]
  []
  [[[:fetch "wο"] [:clear!]]
  [[[:clear!]] [[[:fetch "*QZü"] [:clear!]] [:fetch "α’"]]]
  [[[:size]] [[[:size]] [:delete! "K""]]
  []
  [[[:fetch "t"]]]
  [[[:fetch "$6"] [[[:size]] [[[:size]] [:clear!]]]]
  [[[:fetch "P/7"1"] [:store! "∧=" ""] [:delete! "Â"]
    [:store! "B" "¬É'y"]])}
3. Make 2 "runners"

(defn db-run [db ops] ...)
3. Make 2 “runners”

(defn db-run [db ops]
  (doseq [[op k v] ops]
))
3. Make 2 “runners”

(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      )))
3. Make 2 “runners”

(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :clear! (db/clear! db)
      )))
3. Make 2 "runners"

(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :clear! (db/clear! db)
      )))
3. Make 2 “runners”

(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :clear!  (db/clear!  db)
      :size    (db/size    db)
      )))
3. Make 2 “runners”

(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :clear! (db/clear! db)
      :size   (db/size  db)
      :store! (db/store! db k v)
      :delete! (db/delete! db k)
      :fetch  (db/fetch  db k))))
3. Make 2 “runners”

(defn hm-run [db ops]
  (reduce
   (fn [hm [op k v]]
     
     db ops))
   db ops))
3. Make 2 “runners”

```
(defn hm-run [db ops]
  (reduce
   (fn [hm [op k v]]
     (case op
       ...
       db ops)))
```
3. Make 2 “runners”

(defn hm-run [db ops]
  (reduce
   (fn [hm [op k v]]
     (case op
       :clear! {}))
   db ops))
3. Make 2 “runners”

(defn hm-run [db ops]
  (reduce
   (fn [hm [op k v]]
     (case op
      :clear! {}
      :size hm
      (get hm k v)))
   db ops))
3. Make 2 “runners”

(defn hm-run [db ops]
  (reduce
   (fn [hm [op k v]]
     (case op
       :clear! {}
       :size   hm
       :store! (assoc  hm k v)
     ))
   db ops))
3. Make 2 “runners”

(defn hm-run [db ops]
  (reduce
   (fn [hm [op k v]]
     (case op
      :clear!  {}
      :size    hm
      :store!  (assoc  hm k v)
      :delete! (dissoc hm k)
                      ))
   db ops))
3. Make 2 “runners”

(defn hm-run [db ops]
  (reduce
   (fn [hm [op k v]]
     (case op
      :clear!  {}
      :size    hm
      :store!  (assoc  hm k v)
      :delete! (dissoc hm k)
      :fetch   hm))
   db ops))
4. Define your property

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    ))
4. Define your property

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
           db (db/create)]
      )))


4. Define your property

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
           db (db/create)]
      (db-run db ops)
      )))


4. Define your property

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
          db (db/create)]
      (db-run db ops)
      (equiv? db hm))))
4. Define your property

(defn equiv? [db hm]

  )

(defspec hash-map-eqv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
      db (db/create)]
      (db-run db ops)
      (equiv? db hm)))))
4. Define your property

```
(defn equiv? [db hm]
  (and (= (count hm) (db/size db)))
)

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
           db (db/create)]
      (db-run db ops)
      (equiv? db hm))))
```
4. Define your property

(defn equiv? [db hm]
  (and (= (count hm) (db/size db))
       (every? (fn [[k v]]
                (= v (db/fetch db k)))
               hm)))

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
               (let [hm (hm-run {} ops)
                      db (db/create)]
                 (db-run db ops)
                 (equiv? db hm)))))
Encourage collisions

(def gen-clear (gen/return [:clear!]))
(def gen-size (gen/return [:size]))

(defn gen-store [keys]
  (gen/tuple (gen/return :store!) (gen/elements keys) gen-value))

(defn gen-delete [keys]
  (gen/tuple (gen/return :delete!) (gen/elements keys)))

(defn gen-fetch [keys]
  (gen/tuple (gen/return :fetch) (gen/elements keys)))

(defn gen-ops* [keys]
  (gen/vector
    (gen/one-of [gen-clear
      (gen-store keys)
      (gen-delete keys)
      (gen-fetch keys)
      gen-size]))

(def gen-ops (gen/let [keys (gen/not-empty (gen/vector gen-key))]
  (gen-ops* keys))
> (gen/sample gen-ops)

([]
  []
  [[:fetch "" ] [:size]]
  [[:fetch "w?" ] [:clear!]]
  [[:delete! "Zë" ]]
  [[:fetch "ü" ] [:fetch "ü" ] [:size] [:clear!]]
  []
  [[:store! "Á@m" "k"] [:store! "Á@m" "c"] [:fetch "G," ] [:clear!]]
  [[:clear!]]
  [[:clear!]]
  [[:store! "w§" "ý"] [:clear!] [:fetch "¥p"] [:fetch "♭§"] [:delete! "¥p"]])
> (apply max (map count (gen/sample gen-ops 100)))
91

> (apply max (map count (gen/sample gen-ops 1000)))
96

> (Math/pow 5 91)
4.038967834731581E63
{:result false, :test-var "hash-map-equiv", :seed 1489523387287, :failing-size 26, :num-tests 27, :fail [[[:delete! "9kàµ%|l9PàglDÄF""æy8i)""[:size][[:delete! "\re"U²<\x/Ho^\|ç6lÉ"]][:clear!][[:size][[:clear!][[:delete! "t_©cWuPû"][[:size][[:clear!][[:fetch "B·7{1O]"[:clear!][[:size][[:delete! "¥?t·i\Â\fZ"][[:clear!][[:clear!][[:delete! "È5Pi§uÈVzÔh%êî¿¥W#"]][:size][[:size][[:size][[:delete! "k"]][[:delete! "Me±àÜjÇw 22Yp£~£uiÄQW\tU"]][[:delete! "9kàµ%|l9PàglDÄF""æy8i)"""]][[:delete! "ÈKssy_Îe>"ãµ¶qi3sÀEx¹\f\ʊ\tdb$!"[:size][[:store! "\V%tÇx0û{½%05·zØy\\nàöp$,òUCdGÄB1" "2iôc_ë10!z÷f1">GI?2 13ïÜ<Í0A¸ç"][[:clear!][[:clear!][[:fetch "ë²Ü'ÃÉ·\Y±-Þ(äký2bÇ'FdóN²O£àâh*"][[:delete! "Z'5yúâ'\&v"][[:store! "¥?t·i\Â\fZ" ""][[:store! "ÈKssy_Îe>"ãµ¶qi3sÀEx¹\f\ʊ\tdb$!" "rUmfuRô]2ô'Üb r×úbáÚ¶±9"][[:delete! "öSâRøj/9Èë³ExÎÄ Dx³RB"""'][[:delete! "t_©cWuPû"][[:clear!][[:delete! "@ál%Tuw1°|Çî"][[:store! "IPtDâ\r\"jRÎm]Åü°È\EkÖfÔ<\s\x/ tÜ-BGmOhé" "iÎóf",²-x\batatôyóS\c'\h)ÉºFÖ^"[:fetch "eºòE"][:fetch "t_©cWuPû"[:delete! "(.4")[:clear!][[:fetch "öSâRøj/9Èë³ExÎÄ Dx³RB"""'][[:fetch "ë²Ü'ÃÉ·\Y±-Þ(äký2bÇ'FdóN²O£àâh*"][[:delete! "Z'5yúâ'\&v"][[:size][[:store! "üöyøg°ÎÖv@!ôÖi" "3f>"Sn"][:delete! "OC@Áia"]][[:store! "ÝÜd" "8º$8,QôÚN+,ØkJô!)°ñ44±THA{ôNó=qn\\FÇW"]][[:store! "5BNp9ê3'\TáÔbføQ" "x½Ut×æ#B«GNYX¥taN'tö1""], :shrunken {::total-nodes-visited 361, :depth 162, :result false, :smallest [[[:store! "ø" ""]]]]}}49 operations
But what about race conditions?
Run it in multiple threads

(defn run-in-thread [db ops]
  (.start (Thread. (fn []
    (db-run db ops)))))))

(defn thread-run [db ops-sequences]
  (run! #(run-in-thread db %) ops-sequences))
Wait for them all to finish

(defn run-in-thread [db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
                      (db-run db ops)
                      (deliver done :done!)))))
  done))
(defn run-in-thread [db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
      (db-run db ops)
      (deliver done :done!))))
  done))

(defn thread-run [db ops-sequences]
  (let [threads (map #(run-in-thread db %) ops-sequences)]
    (dorun threads)
    (run! deref threads))
Start all threads at once

(defn run-in-thread [latch db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
                     @latch
                     (db-run db ops)
                     (deliver done :done!)))))
  done))
Start all threads at once

(defn run-in-thread [latch db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
                        @latch
                        (db-run db ops)
                        (deliver done :done!)))]
      done))

(defn thread-run [db ops-sequences]
  (let [latch (promise)
         threads (map #(run-in-thread latch db %) ops-sequences)]
    (dorun threads)
    (deliver latch :go!)
    (run! deref threads)))
Test against the model

(defspec hash-map-equiv 100
)

Test against the model

(defspec hash-map-equiv 100
  (prop/for-all [ops-a gen-ops
                 ops-b gen-ops]
))
Test against the model

(defspec hash-map-equiv 100
  (prop/for-all [ops-a gen-ops
                 ops-b gen-ops]
    (let [ops (concat ops-a ops-b)

    ]

    )))
Test against the model

(defspec hash-map-equiv 100
  (prop/for-all [ops-a gen-ops
                 ops-b gen-ops]
    (let [ops (concat ops-a ops-b)
            hm (hm-run {} ops)
            db (db/create)]
      )))
Test against the model

(defspec hash-map-equiv 100
  (prop/for-all [ops-a gen-ops
                ops-b gen-ops]
    (let [ops (concat ops-a ops-b)
            hm  (hm-run {} ops)
            db  (db/create)]
      (thread-run db [ops-a ops-b])
      (equiv? db hm)))))
Encourage collisions across threads

(defn gen-ops-sequences [n] ...)
Encourage collisions across threads

(defn gen-ops-sequences [n]
  (gen/let [keys (gen/not-empty
                   (gen/vector gen-key))]
    ))
Encourage collisions across threads

(defn gen-ops-sequences [n]
  (gen/let [keys (gen/not-empty
                   (gen/vector gen-key))]
    (apply gen/tuple
           (repeat n (gen-ops* keys))))
Collisions

(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
    (gen-ops-sequences 2)])
  )
)
Collisions

(defspec hash-map-equiv 100
  (prop/for-all [[[ops-a ops-b]]
                (gen-ops-sequences 2)]
    (let [ops (concat ops-a ops-b)
            hm (hm-run {} ops)
            db (db/create)]
      (thread-run db [ops-a ops-b])
      (equiv? db hm))))
Possible interleavings

(defn children [{:keys [sequence threads]}]
  (for [[k [v & thread]] threads]
    {:sequence (conj sequence v)
     :threads (if thread
       (assoc threads k thread)
       (dissoc threads k))})))

(defn branch? [x]
  (-> x :threads not-empty))

(defn possible-interleavings [& sequences]
  (let [threads (into {} (map vector (range) sequences))]
    (->>
      (tree-seq branch? children {:sequence [] :threads threads})
      (remove branch?)
      (map :sequence))))
Equivalent to some possible interleaving

(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
                (gen-ops-sequences 2)]
    (let [ops-i (possible-interleavings ops-a ops-b)
           ]
      )))
Equivalent to some possible interleaving

(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
                (gen-ops-sequences 2)]
    (let [ops-i (possible-interleavings ops-a ops-b)
           db (db/create)]
      (thread-run db [ops-a ops-b]))
    )))
Equivalent to some possible interleaving

(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
    (gen-ops-sequences 2)]
    (let [ops-i (possible-interleavings ops-a ops-b)
      db (db/create)]
      (thread-run db [ops-a ops-b])
      (some? #(equiv? db %)
        (map #(hm-run {} %) ops-i))))))
Repeatability (every?)

(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
    (gen-ops-sequences 2)]
    (let [ops-i (possible-interleavings ops-a ops-b)]
      (every?
        (for [_ (range 10)]
          (let [db (db/create)]
            (thread-run db [ops-a ops-b])
            (some? #(equiv? db %)
              (map #(hm-run {} %) ops-i)))))))
A
[:store! "a" "b"]
[:store! "a" "c"]

B
[:fetch ""]
[:delete! ""]
[:size]
[:delete! ""]
[:fetch ""]
[:fetch ""]
[:size]
[:fetch ""]
...
[:delete! "a"]
Timing

(def gen-sleep (gen/tuple (gen/return :sleep)
                      (gen/choose 1 100)))

(defn gen-ops* [keys]
  (gen/vector
    (gen/one-of [gen-sleep
                 gen-size
                 (gen-fetch keys)
                 (gen-store keys)
                 (gen-delete keys)
                 gen-clear]]))
(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :sleep   (Thread/sleep k)
      :clear!  (db/clear!  db)
      :size    (db/size    db)
      :store!  (db/store!  db k v)
      :delete! (db/delete! db k)
      :fetch   (db/fetch   db k))))
Hash map runner

(defn hm-run [db ops]
  (reduce
   (fn [hm [op k v]]
     (case op
       :sleep   hm
       :clear!  {}
       :size    hm
       :store!  (assoc  hm k v)
       :delete! (dissoc hm k)
       :fetch   hm))
    db ops))
A
[:store! "a" "b"]
[:store! "a" "c"]

B
[:sleep 66]
[:delete! "a"]
Want more?

https://purelyfunctional.tv/stateful-test-check