

State machine modelling and property based testing combined with fault injection

Stevan Andjelkovic

2019.3.22, BOBKonf (Berlin)

Motivation

- ▶ Failures can always happen (network issues, I/O failures, processes killed, etc. . .)
- ▶ Problem: when we test our system (rare) failures usually don't happen
- ▶ In large scale systems rare failures will happen for sure (the law of large numbers)
- ▶ Goal: find bugs related to rare failures before our users!

Simple Testing Can Prevent Most Critical Failures paper (Yuan et al. 2014)

- ▶ The authors studied 198 randomly sampled user-reported failures from five distributed systems (Cassandra, HBase, HDFS, MapReduce, Redis)
- ▶ “Almost all catastrophic failures (48 in total – 92%) are the result of incorrect handling of non-fatal errors explicitly signalled in software.”
- ▶ Example:

```
... } catch (Exception e) {  
    LOG.error(e); // TODO: we should retry here!  
}
```

Overview

- ▶ Property based testing (pure/side-effect free/stateless programs)
- ▶ State machine modelling (monadic/has side-effect/stateful programs)
- ▶ Fault injection (provoking exceptions)
- ▶ Examples using
 - ▶ `quickcheck-state-machine` Haskell library for property based testing
 - ▶ The principles are general and tool independent

Recap: property based testing

- ▶ Unit tests

```
test :: Bool  
test = reverse (reverse [1,2,3]) == [1,2,3]
```

- ▶ Property based tests

```
prop_reverse :: [Int] -> Bool  
prop_reverse xs = reverse (reverse xs) == xs
```

Recap: property based testing

- ▶ Unit tests

```
test :: Bool
test = reverse (reverse [1,2,3]) == [1,2,3]
```

- ▶ Property based tests

```
prop_reverse :: [Int] -> Bool
prop_reverse xs = reverse (reverse xs) == xs
```

- ▶ Running the tests

```
> quickCheck prop_reverse
+++ OK, passed 100 tests.
```

Recap: property based testing (bad)

```
prop_bad :: [Int] -> Bool  
prop_bad xs = reverse xs == xs
```

Recap: property based testing (bad)

```
prop_bad :: [Int] -> Bool
prop_bad xs = reverse xs == xs
```

```
> quickCheck prop_bad
*** Failed! Falsifiable (after 5 tests and 2 shrinks):
[0,1]
[1,0] /= [0,1]
```


State machine modelling (somewhat simplified)

- ▶ Datatype of actions/commands that users can perform
- ▶ A simplified model of the system
- ▶ A transition function explaining how the model evolves for each action
- ▶ Semantics function that executes the action against the real system
- ▶ Post-condition that asserts that the result of execution matches the model

Example: CRUD web application

```
data Action = Create | Read | Update String | Delete
```

```
type Model = Maybe String
```

```
transition :: Model -> Action -> Model
```

```
transition _m Create      = Just ""
```

```
transition m Read         = m
```

```
transition _m (Update s) = Just s
```

```
transition _m Delete      = Nothing
```

Example: CRUD application (continued)

```
data Response = Unit () | String String
```

```
semantics :: Action -> IO Response -- Pseudo code
```

```
semantics Create      = Unit    <$> httpReqPost    url
```

```
semantics Read        = String  <$> httpReqGet      url
```

```
semantics (Update s) = Unit    <$> httpReqPut      url s
```

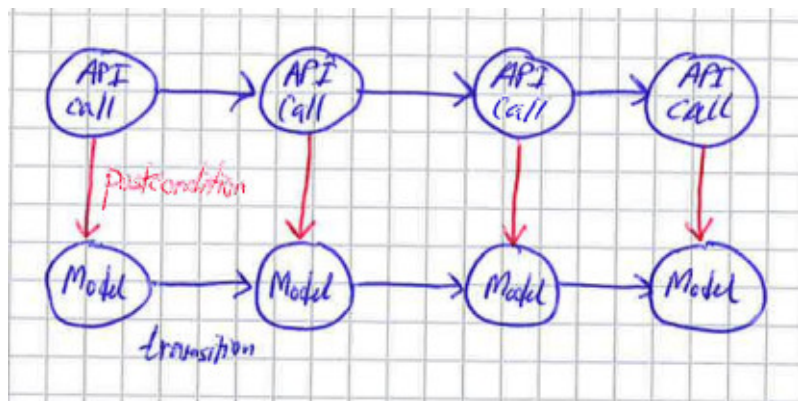
```
semantics Delete      = Unit    <$> httpReqDelete url
```

```
postcondition :: Model -> Action -> Response -> Bool
```

```
postcondition (Just m) Read (String s) = s == m
```

```
postcondition _m      _act _resp      = True
```

State machine modelling as a picture



State machine modelling counterexample

```
> quickCheck prop_stateMachine  
*** Failed! Falsifiable (after 9 tests and 3 shrinks):  
[Create, Update "apa", Read]
```

```
Nothing  
  == Create ==> Unit ()  
Just ""  
  == Update "apa" ==> Unit ()  
Just "apa"  
  == Read ==> String ""  
Just "apa"
```

Postcondition failed for Read, got "" from the system
but expected "apa".

Fault injection

- ▶ Many different tools and libraries, none native to Haskell
- ▶ We'll use the C library `libfiu` (**f**ault **i**njection in **u**space)
- ▶ Two modes of operation
 - ▶ Inject POSIX API/syscall failures
 - ▶ Inject failures at user specified failpoints

Fault injection: syscall failures

- ▶ Using fiu-run directly:

```
fiu-run -x -c 'enable name=posix/io/*' ls
```

- ▶ Via fiu-ctrl in a possibly different process:

```
fiu-run -x top
```

```
fiu-ctrl -c "enable name=posix/io/oc/open" \  
$(pidof top)
```

```
fiu-ctrl -c "disable name=posix/io/oc/open" \  
$(pidof top)
```

Fault injection: user specified failpoints

```
size_t free_space() {
    fiu_return_on("no_free_space", 0);

    [code to find out how much free space there is]
    return space;
}

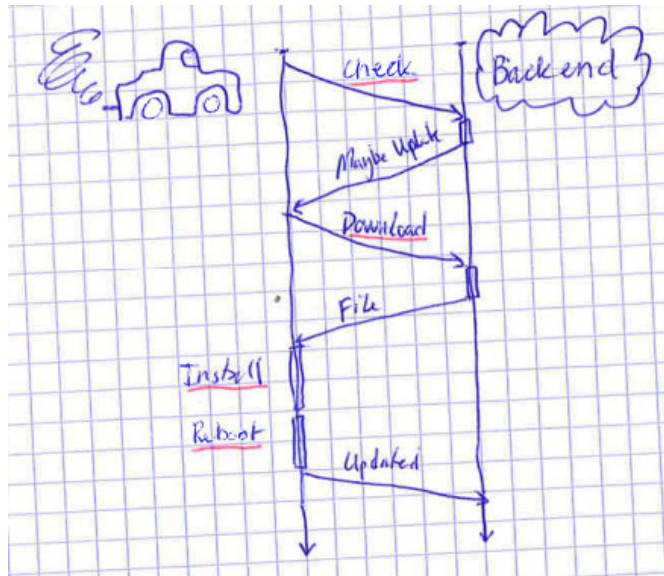
bool file_fits(FILE *fd) {
    if (free_space() < file_size(fd)) {
        return false;
    }
    return true;
}

fiu_init();
fiu_enable("no_free_space", 1, NULL, 0);
assert(file_fits("tmpfile") == false);
```


Examples

- ▶ Over-the-air updates of cars (my workplace)
- ▶ Adjoint Inc's [libraft](#) (consensus algorithm)
- ▶ IOHK's blockchain [database](#)

Over-the-air updates (picture)



Over-the-air updates

```
data Action = Check | Download | Install | Reboot
            | Inject Fault | DisableFaultInject
```

```
data Fault = Network | Kill | GCIOPause | ProcessPrio
           | ReorderReq | SkewClock | RmFile | DamageFile
           | Libfiu (Either Syscall Failpoint)
```

```
inject :: Fault -> IO () -- Pseudo code
inject Network      = call "iptables -j DROP $IP"
inject Kill         = call "kill -9 $PID"
inject GCIOPause    = call "killall -s STOP $PID"
inject ProcessPrio  = call "renice -p $PID"
inject ReorderReq   = call "someproxy" -- Which?
inject SkewClock    = call "faketime $SKEW"
inject RmFile       = call "rm -f $FILE"
inject DamageFile   = call "echo 0 > $FILE"
inject Libfiu       = call "fiu-ctrl -c $FAULT $PID"
```

Over-the-air updates (continued)

```
data Model = Model { fault :: Maybe Fault, ... }

transition :: Model -> Action -> Model
transition m (Fault f)           = m { fault = Just f }
transition m DisableFaultInject = m { fault = Nothing }
transition m ...

postcondition :: Model -> Action -> Response -> Bool
postcondition m act resp = case (fault m, act) of
  (Nothing,      Download) -> resp == Ok
  (Just Network, Download)
    -> resp == TimeoutError
  (Just (Libfiu (Right InstallFailure)), Install)
    -> resp == FailedToInstallError
  ...
```

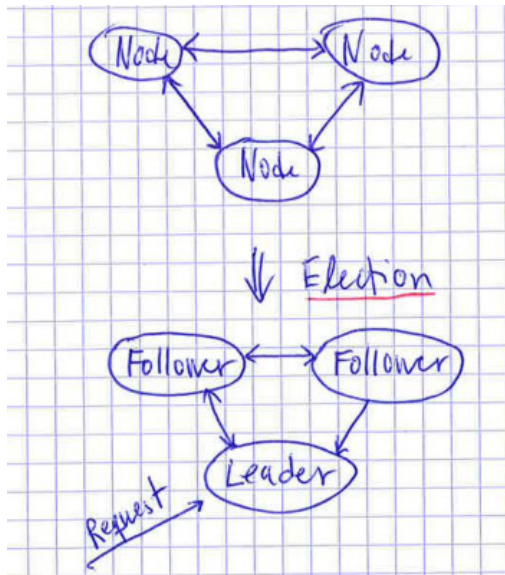
Over-the-air updates (continued 2)

```
prop_reliable :: Property
prop_reliable = forAllActions $ \acts -> do
  (device, update) <- setup
  scheduleUpdateToDevice update device
  let model = initModel { device = Just device }
  (model', result) <- runActions acts model
  assert (result == Success) -- Post-conditions hold
  runActions [ DisableFaultInject
              , Check, Download, Install, Reboot ]
              model'
  update' <- currentUpdate device
  assert (update' == update) -- Always able to recover
```

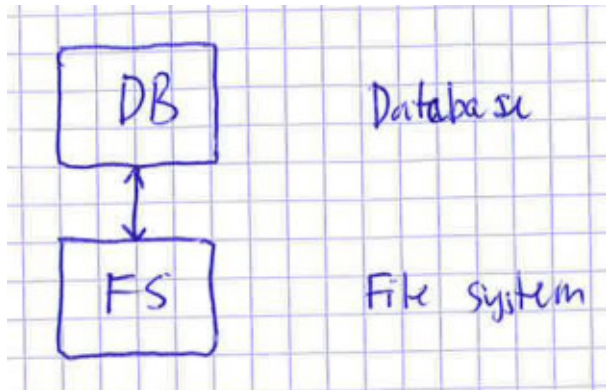
Adjoint Inc's libraft

- ▶ Raft is a consensus algorithm
- ▶ Simplified: how do we get a bunch of nodes to agree on a value?
- ▶ Simplified:
 1. The nodes elect a leader
 2. All requests get forwarded to the leader
 3. Leader makes sure changes get propagated to the followers
- ▶ Complications (faults we inject):
 - ▶ Network traffic loss
 - ▶ Nodes joining and parting
 - ▶ Network partitions
 - ▶ ...

Simplified consensus in pictures



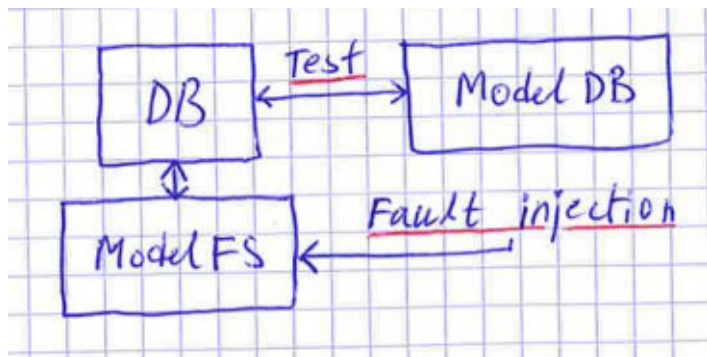
IOHK's blockchain database (picture)



IOHK's blockchain: testing part 1



IOHK's blockchain: testing part 2



Further work

- ▶ Fault injection library for Haskell, c.f.:
 - ▶ FreeBSD's `failpoints`
 - ▶ Rust's `fail-rs` crate
 - ▶ Go's `gofail` library
- ▶ Jepsen-like tests: parallel state machine testing with fault injection and linearisability
- ▶ Lineage-driven fault injection (Alvaro, Rosen, and Hellerstein 2015)

Summary

- ▶ Exceptional circumstances are by definition rare and hence less likely to be tested
- ▶ Exceptional circumstances are often edge cases and hence less likely to be considered when writing the program
- ▶ Exceptional circumstances will nevertheless occur in any long running system
- ▶ By combining fault injection with property based testing we force ourselves to consider these exceptional cases, before our users report them as a bug!

References

- Alvaro, Peter, Joshua Rosen, and Joseph M. Hellerstein. 2015. "Lineage-Driven Fault Injection." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 331–46. SIGMOD '15. New York, NY, USA: ACM. doi:[10.1145/2723372.2723711](https://doi.org/10.1145/2723372.2723711).
- Yuan, Ding, Yu Luo, Xin Zhuang, Renna Guilherme, Xu Rodrigues, Yongle Zhao, Pranay U Zhang, Michael Jain, and Michael Stumm. 2014. "Simple Testing Can Prevent Most Critical Failures — An Analysis of Production Failures in Distributed Data-Intensive Systems," October. doi:[10.13140/2.1.2044.2889](https://doi.org/10.13140/2.1.2044.2889).