# Testing benchmarking and logging

Alexander Diemand Andreas Triantafyllos

November 2018

# Contents

**Abstract**

abstract ...

# Chapter 1

# Testing

## 1.1 Test main entry point

```
module Main
  (
    main
  ) where
import Test.Tasty
import qualified Cardano.BM.Test.Aggregated (tests)
import qualified Cardano.BM.Test.STM (tests)
import qualified Cardano.BM.Test.Trace (tests)
main :: IO ()
main = defaultMain tests
tests :: TestTree
tests =
  testGroup "ouroboros-bm"
  [Cardano.BM.Test ∘ Aggregated.tests
  ,Cardano.BM.Test ∘ STM.tests
  ,Cardano.BM.Test ∘ Trace.tests
  ]
```

### 1.1.1 `instance Arbitrary Aggregated`

```
module Cardano.BM.Arbitrary.Aggregated
where
import Test.QuickCheck
import Cardano.BM.Aggregated
```

We define an instance of *Arbitrary* for an *Aggregated* which lets *QuickCheck* generate arbitrary instances of *Aggregated*. For this an arbitrary list of *Integer* is generated and this list is aggregated into a structure of *Aggregated*.

```
instance Arbitrary Aggregated where
  arbitrary = do
    vs' ← arbitrary :: Gen [Integer]
    let delta as = map (uncurry (-)) $ zip as (tail as)
      sum2 = foldr (λe a → a + e * e) 0
      vs = 42 : 17 : vs'
    return $ Aggregated (Stats (minimum vs) (maximum vs) (toInteger $ length vs) (sum vs) (sum2 vs))
      (last vs)
      (Stats (minimum $ delta vs) (maximum $ delta vs) (toInteger $ length vs) (sum $ delta vs) (sum2 $ delta vs))
```

### 1.1.2 Testing aggregation

```
tests :: TestTree
tests = testGroup "aggregation measurements" [
  property_tests
  ,unit_tests
  ]
property_tests :: TestTree
property_tests = testGroup "Properties" [
  testProperty "minimal" prop_Aggregation_minimal
    ,testProperty "commutative" prop_Aggregation_comm
  ]
unit_tests :: TestTree
unit_tests = testGroup "Unit tests" [
  testCase "initial_minus_1" unit_Aggregation_initial_minus_1
    ,testCase "initial_plus_1" unit_Aggregation_initial_plus_1
    ,testCase "initial_0" unit_Aggregation_initial_zero
  ]
prop_Aggregation_minimal :: Bool
prop_Aggregation_minimal = True
prop_Aggregation_comm :: Integer → Integer → Aggregated → Bool
prop_Aggregation_comm v1 v2 ag =
  let Just (Aggregated stats1 last1 delta1) = updateAggregation v1 $ updateAggregation v2 (Just ag)
    Just (Aggregated stats2 last2 delta2) = updateAggregation v2 $ updateAggregation v1 (Just ag)
  in
  stats1 ≡ stats2 ∧ ((v1 ≡ v2) `implies` (last1 ≡ last2))
    ∧ ((v1 ≡ v2) `implies` (delta1 ≡ delta2))
  -- implication: if p1 is true, then return p2; otherwise true
implies :: Bool → Bool → Bool
implies p1 p2 = (¬ p1) ∨ p2
unit_Aggregation_initial_minus_1 :: Assertion
unit_Aggregation_initial_minus_1 =
  updateAggregation (-1) Nothing @? = Just (Aggregated {
    fstats = Stats (-1) (-1) 1 (-1) 1
    ,flast = (-1)
```

```
      ,fdelta = Stats 0 0 0 0 0})
unit_Aggregation_initial_plus_1 :: Assertion
unit_Aggregation_initial_plus_1 =
  updateAggregation 1 Nothing @? = Just (Aggregated
                                    (Stats 1 1 1 1 1)
                                    1
                                    (Stats 0 0 0 0 0))
unit_Aggregation_initial_zero :: Assertion
unit_Aggregation_initial_zero =
  updateAggregation 0 Nothing @? = Just (Aggregated
                                    (Stats 0 0 1 0 0)
                                    0
                                    (Stats 0 0 0 0 0))
```

### 1.1.3 STM

```
module Cardano.BM.Test.STM (
  tests
  ) where
import Test.Tasty
import Test.Tasty.QuickCheck
tests :: TestTree
tests = testGroup "observing STM actions" [
  testProperty "minimal" prop_STM_observer
    ]
prop_STM_observer :: Bool
prop_STM_observer = True
```

### 1.1.4 Trace

```
tests :: TestTree
tests = testGroup "testing Trace" [
      testProperty "minimal" prop_Trace_minimal
  ,unit_tests
  ,testCase "forked traces stress testing" stress_trace_in_fork
  ,testCase "stress testing: ObservableTrace vs NoTrace" stress_ObservablevsNo_Trace
  ,testCaseInfo "demonstrating nested named context logging" example_named
  ]
unit_tests :: TestTree
unit_tests = testGroup "Unit tests" [
      testCase "opening messages should not be traced" unit_noOpening_Trace
  ,testCase "hierarchy testing" unit_hierarchy
  ,testCase "forked Traces testing" unit_trace_in_fork
  ,testCase "hierarchy testing NoTrace" $
```

```
        unit_hierarchy' [Neutral, NoTrace, (ObservableTrace observablesSet)] onlyLevelOneMessage
    ,testCase "hierarchy testing DropOpening" $
        unit_hierarchy' [Neutral, DropOpening, (ObservableTrace observablesSet)] notObserveOpen
    ,testCase "hierarchy testing UntimedTrace" $
        unit_hierarchy' [Neutral, UntimedTrace, (ObservableTrace observablesSet)] observeOpenWithMeasures
    ,testCase "changing minimum severity at runtime" unit_min_severity
    ,testCase "changing trace-specific severity at runtime" unit_severity_change
    ,testCase "appending names should not exceed 50 chars" uint_append_name
    ]
    where
      observablesSet = fromList [MonotonicClock, MemoryStats]
      notObserveOpen :: [LogObject] → Bool
      notObserveOpen = all (λcase {ObserveOpen _ → False; _ → True})
      onlyLevelOneMessage :: [LogObject] → Bool
      onlyLevelOneMessage = λcase
        [LP (LogMessage (LogItem _ _ "Message from level 1."))] → True
        _ → False
      observeOpenWithMeasures :: [LogObject] → Bool
      observeOpenWithMeasures = any $ λcase
        ObserveOpen (CounterState _ counters) → ¬ $ null counters
        _ → False


prop_Trace_minimal :: Bool
prop_Trace_minimal = True


example_named :: IO String
example_named = do
    logTrace ← setupTrace $ TraceConfiguration StdOut "test" Neutral Debug
    putStrLn "\n"
    logInfo logTrace "entering"
    logTrace0 ← appendName "simple-work-0" logTrace
    complexWork0 logTrace0 "0"
    logTrace1 ← appendName "complex-work-1" logTrace
    complexWork1 logTrace1 "42"
      -- the named context will include "complex" in the logged message
    logInfo logTrace "done."
    return ""
  where
    complexWork0 tr msg = logInfo tr ("let's see: "`append` msg)
    complexWork1 tr msg = do
      logInfo tr ("let's see: "`append` msg)
      logTrace' ← appendName "inner-work-1" tr
      let observablesSet = fromList [MonotonicClock, MemoryStats]
      insertInController logTrace' "STM-action" (ObservableTrace observablesSet)
      _ ← STMObserver.bracketObserveIO logTrace' "STM-action" setVar_
      logInfo logTrace' "let's see: done."
```

```
stress_ObservablevsNo_Trace :: Assertion
stress_ObservablevsNo_Trace = do
    msgs ← STM.newTVarIO [ ]
    trace ← setupTrace $ TraceConfiguration
      (TVarList msgs)
      "test"
      (ObservableTrace (fromList [MonotonicClock]))
      Debug
    msgs' ← STM.newTVarIO [ ]
    trace' ← setupTrace $ TraceConfiguration
      (TVarList msgs')
      "test"
      (ObservableTrace observablesSet)
      Debug
    insertInController trace' "action" (ObservableTrace observablesSet)
    _ ← MonadicObserver.bracketObserveIO trace "" $ observeActions trace' "action"
    res ← STM.readTVarIO msgs
    let endState = findObserveClose res
      startState = findObserveOpen res
      durationObservable = diffTimeObserved startState endState
    putStr ("durationObservable: " ++ show durationObservable ++ "  ")
      -- measurements will not occur
    insertInController trace' "action" NoTrace
    _ ← MonadicObserver.bracketObserveIO trace "" $ observeActions trace' "action"
      -- acquire the traced objects
    res' ← STM.readTVarIO msgs
    let endState' = findObserveClose res'
      startState' = findObserveOpen res'
      durationNoTrace = diffTimeObserved startState' endState'
    putStr ("durationNoTrace: " ++ show durationNoTrace ++ "  ")
      -- time consumed by NoTrace must be lower than ObservableTrace
    assertBool
      ("NoTrace consumed more time than ObservableTrace: " ++ show res')
      (durationNoTrace < durationObservable)
  where
    observablesSet = fromList [MonotonicClock, MemoryStats]
      -- measure 100 times the reversion of a list
    observeActions trace name = do
      forM [1 :: Int .. 100] $ \_ → MonadicObserver.bracketObserveIO trace name action
    action = return $ reverse [1 :: Int .. 1000]
    findObserveClose objects = case find (λcase {(ObserveClose _) → True; _ → False}) objects of
      Just (ObserveClose state) → state
      _ → error "ObserveClose NOT found."
    findObserveOpen objects = case find (λcase {(ObserveOpen _) → True; _ → False}) objects of
      Just (ObserveOpen state) → state
      _ → error "ObserveOpen NOT found."
```

```
unit_hierarchy :: Assertion
unit_hierarchy = do
    msgs ← STM.newTVarIO [ ]
    trace0 ← setupTrace $ TraceConfiguration (TVarList msgs) "test" Neutral Debug
    logInfo trace0 "This should have been displayed!"
      -- subtrace of trace which traces nothing
    insertInController trace0 "inner" NoTrace
    (_,trace1) ← transformTrace "inner" trace0
    logInfo trace1 "This should NOT have been displayed!"
    insertInController trace1 "innest" Neutral
    (_,trace2) ← transformTrace "innest" trace1
    logInfo trace2 "This should NOT have been displayed also due to the trace one level above
      -- acquire the traced objects
    res ← STM.readTVarIO msgs
      -- only the first message should have been traced
    assertBool
      ("Found more or less messages than expected: " ++ show res)
      (length res ≡ 1)

unit_min_severity :: Assertion
unit_min_severity = do
  msgs ← STM.newTVarIO [ ]
  trace ← setupTrace $ TraceConfiguration (TVarList msgs) "test" Neutral Debug
  logInfo trace "Message #1"
  setMinSeverity trace Warning
  logInfo trace "Message #2"
  setMinSeverity trace Info
  logInfo trace "Message #3"
    -- acquire the traced objects
  res ← STM.readTVarIO msgs
    -- only the first message should have been traced
  assertBool
    ("Found Info message when Warning was minimum severity: " ++ show res)
    (all (λcase {(LP (LogMessage (LogItem _ Info "Message #2"))) → False; _ → True}) res)

unit_severity_change :: Assertion
unit_severity_change = do
  msgs ← STM.newTVarIO [ ]
  trace0 ← setupTrace $ TraceConfiguration (TVarList msgs) "test" Neutral Debug
  trace@(ctx, _) ← appendName "sev-change" trace0
  logInfo trace "Message #1"
  setNamedSeverity ctx (loggerName ctx) Warning
  logInfo trace "Message #2"
  setNamedSeverity ctx (loggerName ctx) Info
  logInfo trace "Message #3"
```

```
    -- acquire the traced objects
  res ← STM.readTVarIO msgs
    -- only the first message should have been traced
  assertBool
    ("Found Info message when Warning was minimum severity: " ++ show res)
    (all (λcase {(LP (LogMessage (LogItem _ Info "Message #2"))) → False; _ → True}) res)


unit_hierarchy' :: [TraceTransformer] → ([LogObject] → Bool) → Assertion
unit_hierarchy' (t1 : t2 : t3 : _) f = do
  msgs ← STM.newTVarIO []
  trace1 ← setupTrace $ TraceConfiguration (TVarList msgs) "test" t1 Debug
  logInfo trace1 "Message from level 1."
    -- subtrace of trace which traces nothing
  insertInController trace1 "inner" t2
  (_, trace2) ← transformTrace "inner" trace1
  logInfo trace2 "Message from level 2."
  insertInController trace2 "innest" t3
    -- (_, trace3) ¡- transformTrace "innest" trace2
  _ ← STMObserver.bracketObserveIO trace2 "innest" setVar_
  logInfo trace2 "Message from level 3."
    -- acquire the traced objects
  res ← STM.readTVarIO msgs
    -- only the first message should have been traced
  assertBool
    ("Found more or less messages than expected: " ++ show res)
    (f res)
unit_trace_in_fork :: Assertion
unit_trace_in_fork = do
  msgs ← STM.newTVarIO []
  trace ← setupTrace $ TraceConfiguration (TVarListNamed msgs) "test" Neutral Debug
  trace0 ← appendName "work0" trace
  trace1 ← appendName "work1" trace
  void $ forkIO $ work trace0
  threadDelay 500000
  void $ forkIO $ work trace1
  threadDelay (4 * second)
  res ← STM.readTVarIO msgs
  let names@(_ : namesTail) = map lnName res
    -- each trace should have its own name and log right after the other
  assertBool
    ("Consecutive loggernames are not different: " ++ show names)
    (and $ zipWith (≠) names namesTail)
  where
    work :: Trace IO → IO ()
    work trace = do
      logInfoDelay trace "1"
```

```
    logInfoDelay trace "2"
    logInfoDelay trace "3"
  logInfoDelay :: Trace IO → Text → IO ()
  logInfoDelay trace msg =
    logInfo trace msg ≫
    threadDelay second


stress_trace_in_fork :: Assertion
stress_trace_in_fork = do
  msgs ← STM.newTVarIO [ ]
  trace ← setupTrace $ TraceConfiguration (TVarListNamed msgs) "test" Neutral Debug
  let names = map (λa → ("work-" <> pack (show a))) [1..10]
  forM_ names $ λname → do
    trace' ← appendName name trace
    void $ forkIO $ work trace'
  threadDelay second
  res ← STM.readTVarIO msgs
  let resNames = map lnName res
  let frequencyMap = fromListWith (+) [(x, 1) | x ← resNames]
    -- each trace should have traced 'totalMessages' messages
  assertBool
    ("Frequencies of logged messages according to loggername: " ++ show frequencyMap)
    (all (λname → (lookup ("test." <> name) frequencyMap) ≡ Just totalMessages) names)
 where
  work :: Trace IO → IO ()
  work trace = forM_ [1..totalMessages] $ (logInfo trace) ∘ pack ∘ show
  totalMessages :: Int
  totalMessages = 10


unit_noOpening_Trace :: Assertion
unit_noOpening_Trace = do
  msgs ← STM.newTVarIO [ ]
  logTrace ← setupTrace $ TraceConfiguration (TVarList msgs) "test" DropOpening Debug
  _ ← STMObserver.bracketObserveIO logTrace "setTVar" setVar_
  res ← STM.readTVarIO msgs
  assertBool
    ("Found non-expected ObserveOpen message: " ++ show res)
    (all (λcase {ObserveOpen _ → False; _ → True}) res)


uint_append_name :: Assertion
uint_append_name = do
  trace0 ← setupTrace $ TraceConfiguration StdOut "test" Neutral Debug
  trace1 ← appendName bigName trace0
  (ctx2, _) ← appendName bigName trace1
  assertBool
    ("Found logger name with more than 50 chars: " ++ show (loggerName ctx2))
```

```
        (T.length (loggerName ctx2) ⩽ 50)
    where
      bigName = T.replicate 50 "abcdefghijklmnopqrstuvwxyz"


setVar_ :: STM.STM Integer
setVar_ = do
  t ← STM.newTVar 0
  STM.writeTVar t 42
  res ← STM.readTVar t
  return res
second :: Int
second = 1000000
```