

Testing benchmarking and logging

Alexander Diemand

Andreas Triantafyllos

November 2018

Contents

1	Test coverage	2
1.1	Coverage	2
2	Testing	4
2.1	Test main entry point	4
2.1.1	instance Arbitrary Aggregated	4
2.1.2	Testing aggregation	5
2.1.3	STM	6
2.1.4	Trace	6

Abstract

abstract ...

Chapter 1

Test coverage

1.1 Coverage

Test coverage is calculated as the fraction of functions which are called from test routines. This percentage is calculated by the tool *hpc* with a call to

```
cabal new-test
```

Add to a local `cabal.project.local` file these lines:

```
tests:           True
coverage:        True
library-coverage: True
```

Cardano.BM.Observer.Monadric	100%
Cardano.BM.Data.Trace	100%
Cardano.BM.Counters.Dummy	100%
Cardano.BM.Counters.Common	100%
Cardano.BM.Counters	100%
Cardano.BM.BaseTrace	80%
Cardano.BM.Data.Severity	50%
Cardano.BM.Data.Observable	42%
Cardano.BM.Configuration.Model	41%
Cardano.BM.Data.LogItem	40%
Cardano.BM.Trace	38%
Cardano.BM.Observer.STM	33%
Cardano.BM.Setup	25%
Cardano.BM.Aggregated	23%
Cardano.BM.Data.Counter	22%
Cardano.BM.Output.Switchboard	0%
Cardano.BM.Output.Log	0%
Cardano.BM.Output.EKGView	0%
Cardano.BM.Output.Aggregation	0%
Cardano.BM.Data.SubTrace	0%
Cardano.BM.Data.Rotation	0%
Cardano.BM.Data.Output	0%
Cardano.BM.Data.Backend	0%
Cardano.BM.Configuration	0%
	26%

Figure 1.1: Test coverage of modules in percent as computed by the tool 'hpc'

Chapter 2

Testing

2.1 Test main entry point

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

2.1.1 instance Arbitrary Aggregated

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

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 ( $\lambda e\ a \rightarrow a + e * e$ ) 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))

```

2.1.2 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))

```

2.1.3 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

```

2.1.4 Trace

```

tests :: TestTree
tests = testGroup "testing Trace" [
    unit_tests
    ,testCase "forked traces stress testing" stress_trace_in_fork
    ,testCase "stress testing: ObservableTrace vs. NoTrace" timing_Observable_vs_Untimed
    ,testCaseInfo "demonstrating nested named context logging" example_with_named_contexts
]
unit_tests :: TestTree
unit_tests = testGroup "Unit tests" [
    testCase "opening messages should not be traced" unit_noOpening_Trace
    ,testCase "hierarchy of traces" unit_hierarchy
    ,testCase "forked traces" unit_trace_in_fork
    ,testCase "hierarchy of traces with NoTrace" $
        unit_hierarchy' [Neutral, NoTrace, (ObservableTrace observablesSet)]
]

```



```

    onlyLevelOneMessage
,testCase "hierarchy of traces with DropOpening" $
    unit_hierarchy' [Neutral, DropOpening, (ObservableTrace observablesSet)]
    notObserveOpen
,testCase "hierarchy of traces with UntimedTrace" $
    unit_hierarchy' [Neutral, UntimedTrace, UntimedTrace]
    observeOpenWithoutMeasures
,testCase "changing the minimum severity of a trace at runtime"
    unit_trace_min_severity
,testCase "changing the minimum severity of a named context at runtime"
    unit_named_min_severity
,testCase "appending names should not exceed 50 chars" unit_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
    observeOpenWithoutMeasures :: [LogObject] → Bool
    observeOpenWithoutMeasures = any $ λcase
        ObserveOpen (CounterState _ counters) → null counters
        _ → False

```

Helper routines

```

data TraceConfiguration = TraceConfiguration
    { tcOutputKind :: OutputKind
    , tcName       :: LoggerName
    , tcSubTrace   :: SubTrace
    , tcSeverity   :: Severity
    }

setupTrace :: TraceConfiguration → IO (Trace IO)
setupTrace (TraceConfiguration outk name trafo sev) = do
    c ← liftIO $ Cardano.BM.Configuration.setup "some_file_path.yaml"
    ctx ← liftIO $ newContext name c sev
    let logTrace0 = case outk of
        StdOut → BaseTrace.natTrace liftIO stdoutTrace
        TVarList tvar → BaseTrace.natTrace liftIO $ traceInTVarIO tvar
        TVarListNamed tvar → BaseTrace.natTrace liftIO $ traceNamedInTVarIO tvar
        Null → noTrace
    setSubTrace (configuration ctx) name (Just trafo)
    logTrace' ← subTrace "" (ctx, logTrace0)
    return logTrace'

```

```

setTransformer_ :: Trace IO → LoggerName → Maybe SubTrace → IO ()
setTransformer_ (ctx, _) name subtr = do
  let c = configuration ctx
      n = (loggerName ctx) <> " ." <> name
  setSubTrace c n subtr

setMinSeverity_ :: Configuration → Severity → IO ()
setMinSeverity_ c s = do
  setMinSeverity c s

setNamedSeverity_ :: Configuration → LoggerName → Severity → IO ()
setNamedSeverity_ c n s = do
  setSeverity c n (Just s)

```

Example of using named contexts with *Trace*

```

example_with_named_contexts :: IO String
example_with_named_contexts = 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 (0): " 'append' msg)
  complexWork1 tr msg = do
    logInfo tr ("let's see (1): " 'append' msg)
    logTrace' ← appendName "inner-work-1" tr
    let observablesSet = fromList [MonotonicClock, MemoryStats]
    setTransformer_logTrace' "STM-action" (Just $ ObservableTrace observablesSet)
    _ ← STMObserver.bracketObserveIO logTrace' "STM-action" setVar_
    logInfo logTrace' "let's see: done."

```

Show effect of turning off observables

```

run_timed_action :: Trace IO → IO (Microsecond)
run_timed_action logTrace = do
  runid ← newUnique
  t0 ← getMonoClock
  _ ← observeAction logTrace "Observables"
  t1 ← getMonoClock
  return $ diffTimeObserved (CounterState runid t0) (CounterState runid t1)

```

```

where
  observeAction trace name = do
    _ ← MonadicObserver.bracketObserveIO trace name action
    return ()
  action = return $ forM [1 :: Int..100] $ \_ → reverse [1 :: Int..1000]
timing_Observable_vs_Untimed :: Assertion
timing_Observable_vs_Untimed = do
  msgs1 ← STM.newTVarIO []
  trace1 ← setupTrace $ TraceConfiguration
    (TVarList msgs1)
    "observables"
    (ObservableTrace observablesSet)
    Debug
  msgs2 ← STM.newTVarIO []
  trace2 ← setupTrace $ TraceConfiguration
    (TVarList msgs2)
    "no timing"
    UntimedTrace
    Debug
  msgs3 ← STM.newTVarIO []
  trace3 ← setupTrace $ TraceConfiguration
    (TVarList msgs3)
    "no trace"
    NoTrace
    Debug
  t_observable ← run_timed_action trace1
  t_untimed ← run_timed_action trace2
  t_notrace ← run_timed_action trace3
  assertBool
    ("Untimed consumed more time than ObservableTrace " ++ (show [t_untimed,t_observable]))
    (t_untimed < t_observable)
  assertBool
    ("NoTrace consumed more time than ObservableTrace" ++ (show [t_notrace,t_observable]))
    (t_notrace < t_observable)
  assertBool
    ("NoTrace consumed more time than Untimed" ++ (show [t_notrace,t_untimed]))
    True
where
  observablesSet = fromList [MonotonicClock,MemoryStats]

```

Control tracing in a hierarchy of Traces

We can lay out traces in a hierarchical manner, that the children forward traced items to the parent *Trace*. A *NoTrace* introduced in this hierarchy will cut off a branch from messaging to the root.

```

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
  setTransformer_trace0 "inner" (Just NoTrace)
  trace1 ← subTrace "inner" trace0
  logInfo trace1 "This should NOT have been displayed!"
  setTransformer_trace1 "innermost" (Just Neutral)
  trace2 ← subTrace "innermost" 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)

```

Change a trace's minimum severity

A trace is configured with a minimum severity and filters out messages that are labelled with a lower severity. This minimum severity of the current trace can be changed.

```

unit_trace_min_severity::Assertion
unit_trace_min_severity = do
  msgs ← STM.newTVarIO []
  trace@(ctx, _) ← setupTrace $ TraceConfiguration (TVarList msgs) "test min severity" Neutral Debug
  logInfo trace "Message #1"
  -- raise the minimum severity to Warning
  setMinSeverity_(configuration ctx) Warning
  msev ← Cardano.BM.Configuration.minSeverity (configuration ctx)
  assertBool ("min severity should be Warning, but is " ++ (show msev))
    (msev == Warning)
  -- this message will not be traced
  logInfo trace "Message #2"
  -- lower the minimum severity to Info
  setMinSeverity_(configuration ctx) Info
  -- this message is traced
  logInfo trace "Message #3"
  -- acquire the traced objects
  res ← STM.readTVarIO msgs
  -- only the first and last messages should have been traced
  assertBool
    ("Found more or less messages than expected: " ++ show res)

```

```

    (length res == 2)
  assertBool
    ("Found Info message when Warning was minimum severity: " ++ show res)
    (all (\case {(LP (LogMessage (LogItem _ Info "Message #2"))} → False; _ → True)}) res)

```

Change the minimum severity of a named context

A trace of a named context can be configured with a minimum severity, such that the trace will filter out messages that are labelled with a lower severity.

```

unit_named_min_severity :: Assertion
unit_named_min_severity = do
  msgs ← STM.newTVarIO []
  trace0 ← setupTrace $ TraceConfiguration (TVarList msgs) "test named severity" Neutral Debug
  trace@(ctx, _) ← appendName "sev-change" trace0
  logInfo trace "Message #1"

  -- raise the minimum severity to Warning
  setNamedSeverity_ (configuration ctx) (loggerName ctx) Warning
  msev ← Cardano.BM.Configuration.inspectSeverity (configuration ctx) (loggerName ctx)
  assertBool ("min severity should be Warning, but is " ++ (show msev))
    (msev == Just Warning)
  -- this message will not be traced
  logInfo trace "Message #2"

  -- lower the minimum severity to Info
  setNamedSeverity_ (configuration ctx) (loggerName ctx) Info
  -- this message is traced
  logInfo trace "Message #3"

  -- acquire the traced objects
  res ← STM.readTVarIO msgs
  -- only the first and last messages should have been traced
  assertBool
    ("Found more or less messages than expected: " ++ show res)
    (length res == 2)
  assertBool
    ("Found Info message when Warning was minimum severity: " ++ show res)
    (all (\case {(LP (LogMessage (LogItem _ Info "Message #2"))} → False; _ → True)}) res)

unit_hierarchy' :: [SubTrace] → ([LogObject] → Bool) → Assertion
unit_hierarchy' subtraces f = do
  let (t1:t2:t3:_) = cycle subtraces
  msgs ← STM.newTVarIO []
  -- create trace of type 1
  trace1 ← setupTrace $ TraceConfiguration (TVarList msgs) "test" t1 Debug
  logInfo trace1 "Message from level 1."

  -- subtrace of type 2
  setTransformer trace1 "inner" (Just t2)

```

```

trace2 ← subTrace "inner" trace1
logInfo trace2 "Message from level 2."
  -- subsubtrace of type 3
setTransformer trace2 "innermost" (Just t3)
_ ← STMObserver.bracketObserveIO trace2 "innermost" 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 logername: " ++ 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)

```

Assert maximum length of log context name

The name of the log context cannot grow beyond a maximum number of characters, currently the limit is set to 50.

```

unit_append_name :: Assertion
unit_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 "abcdefghijklmnopqrstuvwxy"

setVar_ :: STM.STM Integer
setVar_ = do
  t ← STM.newTVar 0
  STM.writeTVar t 42

```

```
    res ← STM.readTVar t  
    return res  
second :: Int  
second = 1000000
```