Testing benchmarking and logging

Alexander Diemand   Andreas Triantafyllos

November 2018

Contents

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.Data.Trace	100%
Cardano.BM.Counters.Dummy	100%
Cardano.BM.Counters	100%
Cardano.BM.Configuration	100%
Cardano.BM.Output.Switchboard	90%
Cardano.BM.Data.Configuration	83%
Cardano.BM.BaseTrace	80%
Cardano.BM.Configuration.Model	77%
Cardano.BM.Setup	75%
Cardano.BM.Observer.Monadic	75%
Cardano.BM.Output.Log	66%
Cardano.BM.Counters.Common	66%
Cardano.BM.Data.Aggregated	64%
Cardano.BM.Data.Counter	56%
Cardano.BM.Data.Severity	54%
Cardano.BM.Data.Output	50%
Cardano.BM.Data.BackendKind	50%
Cardano.BM.Data.Backend	50%
Cardano.BM.Configuration.Static	50%
Cardano.BM.Output.Aggregation	42%
Cardano.BM.Data.LogItem	40%
Cardano.BM.Observer.STM	33%
Cardano.BM.Data.AggregatedKind	33%
Cardano.BM.Trace	31%
Cardano.BM.Data.Observable	20%
Cardano.BM.Data.SubTrace	13%
Cardano.BM.Data.Rotation	10%
Cardano.BM.Output.EKGView	0%
Paths_iohk_monitoring	0%
	50%

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.Aggregated (tests)
import qualified Cardano.BM.Test.STM (tests)
import qualified Cardano.BM.Test.Trace (tests)
import qualified Cardano.BM.Test.Configuration (tests)
import qualified Cardano.BM.Test.Routing (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
  , Cardano.BM.Test ∘ Configuration.tests
  , Cardano.BM.Test ∘ Routing.tests
  ]
```

2.1.1 instance Arbitrary 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]
```

4

```
    let vs = 42 : 17 : vs'
        (m, s) = updateMeanVar $ map fromInteger vs
    return $ AggregatedStats (Stats
        (PureI (last vs))
        (PureI (minimum vs))
        (PureI (maximum vs))
        (toInteger $ length vs)
        (m)
        (s)
        )
```

Estimators for mean and variance must be updated the same way as in the code.

```
updateMeanVar :: [Double] → (Double, Double)
updateMeanVar [ ] = (0, 0)
updateMeanVar (val : vals) = updateMeanVar' (val, 0) 1 vals
    where
        updateMeanVar' (m, s) _ [ ] = (m, s)
        updateMeanVar' (m, s) cnt (a : r) =
            let delta = a − m
                newcount = cnt + 1
                m' = m + (delta / newcount)
                s' = s + (delta * (a − m'))
            in
            updateMeanVar' (m', s') newcount r
```

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 AggregatedStats stats1 = updateAggregation (PureI v1) (updateAggregation (PureI v2) ag Nothing) Nothing
      AggregatedStats stats2 = updateAggregation (PureI v2) (updateAggregation (PureI v1) ag Nothing) Nothing
  in
  fmin stats1 ≡ fmin stats2 ∧
  fmax stats1 ≡ fmax stats2 ∧
  fcount stats1 ≡ fcount stats2 ∧
  abs (fsum_A stats1 − fsum_A stats2) < 1.0e−4 ∧
  abs (fsum_B stats1 − fsum_B stats2) < 1.0e−4 ∧
  (v1 ≡ v2) 'implies' (flast stats1 ≡ flast stats2)
-- 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) firstStateAggregatedStats Nothing @? =
    AggregatedStats (Stats (−1) (−1) 0 2 (−0.5) 0.5)
unit_Aggregation_initial_plus_1 :: Assertion
unit_Aggregation_initial_plus_1 =
  updateAggregation 1 firstStateAggregatedStats Nothing @? = AggregatedStats (Stats 1 0 1 2 0.5 0.5)
unit_Aggregation_initial_zero :: Assertion
unit_Aggregation_initial_zero =
  updateAggregation 0 firstStateAggregatedStats Nothing @? = AggregatedStats (Stats 0 0 0 2 0 0)

firstStateAggregatedStats :: Aggregated
firstStateAggregatedStats = AggregatedStats (Stats 0 0 0 1 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 80 chars" unit_append_name
    ,testCase "creat subtrace which duplicates messages" unit_trace_duplicate
    ,testCase "testing name filtering" unit_name_filtering
    ]
    where
      observablesSet = [MonotonicClock,MemoryStats]
      notObserveOpen :: [LogObject] → Bool
      notObserveOpen = all (λcase {ObserveOpen _ → False; _ → True})
      onlyLevelOneMessage :: [LogObject] → Bool
      onlyLevelOneMessage = λcase
        [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 subTr sev) = do
    c ← liftIO $ Cardano.BM.Configuration ◦ Model.empty
    mockSwitchboard ← newMVar $ error "Switchboard uninitialized."
    ctx ← liftIO $ newContext name c sev $ Switchboard mockSwitchboard
    let logTrace0 = case outk of
        TVarList tvar → BaseTrace.natTrace liftIO $ traceInTVarIO tvar
        TVarListNamed tvar → BaseTrace.natTrace liftIO $ traceNamedInTVarIO tvar
    setSubTrace (configuration ctx) name (Just subTr)
    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
```

Example of using named contexts with *Trace*

```
example_with_named_contexts :: IO String
example_with_named_contexts = do
    cfg ← defaultConfigTesting
    logTrace ← Setup.setupTrace (Right cfg) "test"
    putStrLn "\n"
    logInfo logTrace "entering"
    logTrace0 ← appendName "simple-work-0" logTrace
    work0 ← complexWork0 logTrace0 "0"
    logTrace1 ← appendName "complex-work-1" logTrace
    work1 ← complexWork1 logTrace1 "42"

    Async.wait work0
    Async.wait work1
    -- the named context will include "complex" in the logged message
    logInfo logTrace "done."
    threadDelay 1000
    return ""
  where
    complexWork0 tr msg = Async.async $ logInfo tr ("let's see (0): " `append` msg)
    complexWork1 tr msg = Async.async $ do
        logInfo tr ("let's see (1): " `append` msg)
        trInner@(ctx, _) ← appendName "inner-work-1" tr
        let observablesSet = [MonotonicClock]
        setSubTrace (configuration ctx) "test.complex-work-1.inner-work-1.STM-action" $
            Just $ ObservableTrace observablesSet
        _ ← STMObserver.bracketObserveIO trInner "STM-action" setVar_
```

```
        logInfo trInner "let's see: done."
        -- 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*
    *forM_* [(1 :: *Int*)..10] $ *const* $ *observeAction logTrace*
    t1 ← *getMonoClock*
    return $ *diffTimeObserved* (*CounterState runid t0*) (*CounterState runid t1*)
  where
    observeAction trace = **do**
      _ ← *MonadicObserver.bracketObserveIO trace* "" *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* [ ]
    traceObservable ← *setupTrace* $ *TraceConfiguration*
      (*TVarList msgs1*)
      "observables"
      (*ObservableTrace observablesSet*)
      Debug
    msgs2 ← *STM.newTVarIO* [ ]
    traceUntimed ← *setupTrace* $ *TraceConfiguration*
      (*TVarList msgs2*)
      "no timing"
      UntimedTrace
      Debug
    msgs3 ← *STM.newTVarIO* [ ]
    traceNoTrace ← *setupTrace* $ *TraceConfiguration*
      (*TVarList msgs3*)
      "no trace"
      NoTrace
      Debug
    t_observable ← *run_timed_action traceObservable*
    t_untimed ← *run_timed_action traceUntimed*
    t_notrace ← *run_timed_action traceNoTrace*
    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 = [*MonotonicClock*, *MemoryStats*]

Control tracing in a hierarchy of *Trace*s

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 la-
belled 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* (λ**case** {(*LogMessage* (*LogItem* _ *Info* `"Message #2"`)) → *False*; _ → *True*}) *res*)

Define a subtrace's behaviour to duplicate all messages

The *SubTrace* will duplicate all messages that pass through it. Each message will be
in its own named context.

```
unit_trace_duplicate :: Assertion
```
`unit_trace_duplicate` = **do**
  msgs ← *STM.newTVarIO* [ ]
  trace0@(*ctx*, _) ← *setupTrace* $ *TraceConfiguration* (*TVarList msgs*) `"test duplicate"` *Neutral Debug*
  logInfo trace0 `"Message #1"`

  ```
  -- create a subtrace which duplicates all messages
  ```
  setSubTrace (*configuration ctx*) `"test duplicate.orig"` $ *Just* (*TeeTrace* `"dup"`)
  trace ← *subTrace* `"orig"` *trace0*

  ```
  -- this message will be duplicated
  ```
  logInfo trace `"You will see me twice!"`

  ```
  -- 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* ≡ 3)

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
    setSeverity (configuration ctx) (loggerName ctx) (Just 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
    setSeverity (configuration ctx) (loggerName ctx) (Just 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 (λcase {(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)
```

Logging in parallel

```
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
    work0 ← work trace0
    threadDelay 5000
    work1 ← work trace1
    Async.wait $ work0
    Async.wait $ work1

    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 (Async.Async ())
    work trace = Async.async $ do
        logInfoDelay trace "1"
        logInfoDelay trace "2"
        logInfoDelay trace "3"
    logInfoDelay :: Trace IO → Text → IO ()
    logInfoDelay trace msg =
        logInfo trace msg ≫
        threadDelay 10000
```

Stress testing parallel logging

```
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 :: Int)]
    ts ← forM names $ λname → do
        trace' ← appendName name trace
        work trace'
    forM_ ts Async.wait

    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 (Async.Async ())
    work trace = Async.async $ forM_ [1 .. totalMessages] $ (logInfo trace) ∘ pack ∘ show
    totalMessages :: Int
    totalMessages = 10
```

Dropping *ObserveOpen* **messages in a subtrace**

```
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, cur-
rently the limit is set to 80.

```
unit_append_name :: Assertion
unit_append_name = do
  cfg ← defaultConfigTesting
  trace0 ← Setup.setupTrace (Right cfg) "test"
  trace1 ← appendName bigName trace0
  (ctx2, _) ← appendName bigName trace1
  assertBool
    ("Found logger name with more than 80 chars: " ++ show (loggerName ctx2))
    (T.length (loggerName ctx2) ⩽ 80)
  where
    bigName = T.replicate 30 "abcdefghijklmnopqrstuvwxyz"
```

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

Testing log context name filters

unit_name_filtering :: *Assertion*
unit_name_filtering = **do**
  let *contextName* = `"test.sub.1"`
  let *loname* = `"sum"`-- would be part of a `"LogValue loname 42"`

  let *filter1* = [*Drop* (*Exact* `"test.sub.1"`)]
  assertBool (`"Dropping a specific name should filter it out and thus return False"`)
    (*False* ≡ *evalFilters filter1 contextName*)
  let *filter2* = [*Drop* (*EndsWith* `".1"`)]
  assertBool (`"Dropping a name ending with a specific text should filter out the context nam`
    (*False* ≡ *evalFilters filter2 contextName*)
  let *filter3* = [*Drop* (*StartsWith* `"test."`)]
  assertBool (`"Dropping a name starting with a specific text should filter out the context n`
    (*False* ≡ *evalFilters filter3 contextName*)
  let *filter4* = [*Drop* (*Contains* `".sub."`)]
  assertBool (`"Dropping a name starting containing a specific text should filter out the con`
    (*False* ≡ *evalFilters filter4 contextName*)
  let *filter5* = [*Drop* (*StartsWith* `"test."`),
    Unhide (*Exact* `"test.sub.1"`)]
  assertBool (`"Dropping all and unhiding a specific name should the context name allow passi`
    (*True* ≡ *evalFilters filter5 contextName*)
  let *filter6* = [*Drop* (*StartsWith* `"test."`),
    Unhide (*EndsWith* `".sum"`)]
  assertBool (`"Dropping all and unhiding a named value, the LogObject should pass the filter`
    (*True* ≡ *evalFilters filter6* (*contextName* <> `"."` <> *loname*))
  let *filter7* = [*Drop* (*StartsWith* `"test."`),
    Unhide (*EndsWith* `".product"`)]
  assertBool (`"Dropping all and unhiding an inexistant named value, the LogObject should not`
    (*False* ≡ *evalFilters filter7* (*contextName* <> `"."` <> *loname*))

2.1.5 Testing configuration

Test declarations

tests :: *TestTree*
tests = *testGroup* `"config tests"` [
  property_tests
  ,*unit_tests*
  ]
property_tests :: *TestTree*
property_tests = *testGroup* `"Properties"` [
  testProperty `"minimal"` *prop_Configuration_minimal*
  ]
unit_tests :: *TestTree*
unit_tests = *testGroup* `"Unit tests"` [

```
  testCase "static_representation" unit_Configuration_static_representation
 ,testCase "parsed_representation" unit_Configuration_parsed_representation
 ,testCase "parsed_configuration" unit_Configuration_parsed
 ,testCase "include_EKG_if_defined" unit_Configuration_check_EKG_positive
 ,testCase "not_include_EKG_if_ndef" unit_Configuration_check_EKG_negative
 ]
```

Property tests

```
prop_Configuration_minimal :: Bool
prop_Configuration_minimal = True
```

Unit tests

The configuration file only indicates that EKG is listening on port nnnnn. Infer
that *EKGViewBK* needs to be started as a backend.

```
unit_Configuration_check_EKG_positive :: Assertion
unit_Configuration_check_EKG_positive = do
  let c = ["rotation:"
    ," rpLogLimitBytes: 5000000"
    ," rpKeepFilesNum: 10"
    ," rpMaxAgeHours: 24"
    ,"minSeverity: Info"
    ,"defaultBackends:"
    ," - KatipBK"
    ,"setupBackends:"
    ," - KatipBK"
    ,"defaultScribes:"
    ,"- - StdoutSK"
    ," - stdout"
    ,"setupScribes:"
    ,"- scName: stdout"
    ," scRotation: null"
    ," scKind: StdoutSK"
    ,"hasEKG: 18321"
    ,"options:"
    ," test:"
    ," value: nothing"
    ]
   fp = "/tmp/test_ekgv_config.yaml"
  writeFile fp $ unlines c
  repr ← parseRepresentation fp
  assertBool "expecting EKGViewBK to be setup" $
    EKGViewBK ∈ (setupBackends repr)
```

If there is no port defined for `EKG`, then do not start it even if present in the config.

 unit_Configuration_check_EKG_negative :: *Assertion*
 unit_Configuration_check_EKG_negative = **do**
  let *c* = [`"rotation:"`
   ,`"   rpLogLimitBytes: 5000000"`
   ,`"   rpKeepFilesNum: 10"`
   ,`"   rpMaxAgeHours: 24"`
   ,`"minSeverity: Info"`
   ,`"defaultBackends:"`
   ,`"   - KatipBK"`
   ,`"   - EKGViewBK"`
   ,`"setupBackends:"`
   ,`"   - KatipBK"`
   ,`"   - EKGViewBK"`
   ,`"defaultScribes:"`
   ,`"- - StdoutSK"`
   ,`"   - stdout"`
   ,`"setupScribes:"`
   ,`"- scName: stdout"`
   ,`"   scRotation: null"`
   ,`"   scKind: StdoutSK"`
   ,`"###hasEKG: 18321"`
   ,`"options:"`
   ,`"   test:"`
   ,`"     value: nothing"`
   ]
   fp = `"/tmp/test_ekgv_config.yaml"`
  writeFile fp $ *unlines c*
  repr ← *parseRepresentation fp*

  assertBool `"EKGViewBK shall not be setup"` $
   ¬ $ *EKGViewBK* ∈ (*setupBackends repr*)
  assertBool `"EKGViewBK shall not receive messages"` $
   ¬ $ *EKGViewBK* ∈ (*defaultBackends repr*)

 unit_Configuration_static_representation :: *Assertion*
 unit_Configuration_static_representation =
  let *r* = *Representation*
    {*minSeverity* = *Info*
    ,*rotation* = *RotationParameters* 5000000 24 10
    ,*setupScribes* =
     [*ScribeDefinition* {*scName* = `"stdout"`
         ,*scKind* = *StdoutSK*
         ,*scRotation* = *Nothing*}
     ]
    ,*defaultScribes* = [(*StdoutSK*, `"stdout"`)]
    ,*setupBackends* = [*EKGViewBK*, *KatipBK*]

```
        ,defaultBackends = [KatipBK]
        ,hasGUI = Just 12789
        ,hasEKG = Just 18321
        ,options =
          HM.fromList [("test1",(HM.singleton "value" "object1"))
             ,("test2",(HM.singleton "value" "object2"))]
        }
    in
    encode r @? = ""
"rotation:\n"
"   rpLogLimitBytes: 5000000\n"
"   rpKeepFilesNum: 10\n"
"   rpMaxAgeHours: 24\n"
"defaultBackends:\n"
"- KatipBK\n"
"setupBackends:\n"
"- EKGViewBK\n"
"- KatipBK\n"
"hasGUI: 12789\n"
"defaultScribes:\n"
"- - StdoutSK\n"
"  - stdout\n"
"options:\n"
"  test2:\n"
"    value: object2\n"
"  test1:\n"
"    value: object1\n"
"setupScribes:\n"
"- scName: stdout\n"
"  scRotation: null\n"
"  scKind: StdoutSK\n"
"hasEKG: 18321\n"
"minSeverity: Info\n"

unit_Configuration_parsed_representation :: Assertion
unit_Configuration_parsed_representation = do
   repr ← parseRepresentation "test/config.yaml"
   encode repr @? = ""
"rotation:\n"
"   rpLogLimitBytes: 5000000\n"
"   rpKeepFilesNum: 10\n"
"   rpMaxAgeHours: 24\n"
"defaultBackends:\n"
"- KatipBK\n"
"setupBackends:\n"
"- AggregationBK\n"
"- EKGViewBK\n"
"- KatipBK\n"
```

```
"hasGUI: null\n"
"defaultScribes:\n"
"- - StdoutSK\n"
"  - stdout\n"
"options:\n"
"  mapSubtrace:\n"
"    iohk.benchmarking:\n"
"      tag: ObservableTrace\n"
"      contents:\n"
"      - GhcRtsStats\n"
"      - MonotonicClock\n"
"    iohk.deadend: NoTrace\n"
"  mapSeverity:\n"
"    iohk.startup: Debug\n"
"    iohk.background.process: Error\n"
"    iohk.testing.uncritical: Warning\n"
"  mapAggregatedkinds:\n"
"    iohk.interesting.value: EwmaAK {alpha = 0.75}\n"
"    iohk.background.process: StatsAK\n"
"  cfokey:\n"
"    value: Release-1.0.0\n"
"  mapScribes:\n"
"    iohk.interesting.value:\n"
"    - StdoutSK::stdout\n"
"    - FileTextSK::testlog\n"
"    iohk.background.process: FileTextSK::testlog\n"
"  mapBackends:\n"
"    iohk.interesting.value:\n"
"    - EKGViewBK\n"
"    - AggregationBK\n"
"setupScribes:\n"
"- scName: testlog\n"
"  scRotation:\n"
"    rpLogLimitBytes: 25000000\n"
"    rpKeepFilesNum: 3\n"
"    rpMaxAgeHours: 24\n"
"  scKind: FileTextSK\n"
"- scName: stdout\n"
"  scRotation: null\n"
"  scKind: StdoutSK\n"
"hasEKG: 12789\n"
"minSeverity: Info\n"
```

unit_Configuration_parsed :: *Assertion*
unit_Configuration_parsed = **do**
  cfg ← *setup* `"test/config.yaml"`
  cfgInternal ← *readMVar* $ *getCG cfg*
  cfgInternal @? = *ConfigurationInternal*

```
{cgMinSeverity    = Info
,cgMapSeverity    = HM.fromList [("iohk.startup",Debug)
                    ,("iohk.background.process",Error)
                    ,("iohk.testing.uncritical",Warning)
                    ]
,cgMapSubtrace    = HM.fromList [("iohk.benchmarking",
                          ObservableTrace [GhcRtsStats,MonotonicClock])
                    ,("iohk.deadend",NoTrace)
                    ]
,cgOptions        = HM.fromList
  [("mapSubtrace",
    HM.fromList [("iohk.benchmarking",
                Object (HM.fromList [("tag",String "ObservableTrace")
                ,("contents",Array $ V.fromList
                            [String "GhcRtsStats"
                            ,String "MonotonicClock"])]))
      ,("iohk.deadend",String "NoTrace")])
  ,("mapSeverity",HM.fromList [("iohk.startup",String "Debug")
    ,("iohk.background.process",String "Error")
    ,("iohk.testing.uncritical",String "Warning")])
  ,("mapAggregatedkinds",HM.fromList [("iohk.interesting.value",
                            String "EwmaAK {alpha = 0.75}")
                          ,("iohk.background.process",
                            String "StatsAK")])
  ,("cfokey",HM.fromList [("value",String "Release-1.0.0")])
  ,("mapScribes",HM.fromList [("iohk.interesting.value",
                Array $ V.fromList [String "StdoutSK::stdout"
                  ,String "FileTextSK::testlog"])
    ,("iohk.background.process",String "FileTextSK::testlog")])
  ,("mapBackends",HM.fromList [("iohk.interesting.value",
    Array $ V.fromList [String "EKGViewBK"
      ,String "AggregationBK"])])
  ]
,cgMapBackend     = HM.fromList [("iohk.interesting.value",[EKGViewBK,AggregationBK])]
,cgDefBackendKs   = [KatipBK]
,cgSetupBackends  = [AggregationBK,EKGViewBK,KatipBK]
,cgMapScribe      = HM.fromList [("iohk.interesting.value",
                    ["StdoutSK::stdout","FileTextSK::testlog"])
                  ,("iohk.background.process",["FileTextSK::testlog"])
                  ]
,cgDefScribes     = ["StdoutSK::stdout"]
,cgSetupScribes   = [ScribeDefinition
                      {scKind   = FileTextSK
                      ,scName  = "testlog"
                      ,scRotation = Just $ RotationParameters
                        {rpLogLimitBytes = 25000000
                        ,rpMaxAgeHours = 24
                        ,rpKeepFilesNum = 3
```

```
                                }
                              }
                     ,ScribeDefinition
                        {scKind = StdoutSK
                        ,scName = "stdout"
                        ,scRotation = Nothing
                        }
                    ]
  ,cgMapAggregatedKind = HM.fromList [("iohk.interesting.value",EwmaAK {alpha = 0.75})
                    ,("iohk.background.process",StatsAK)
                    ]
  ,cgDefAggregatedKind = StatsAK
  ,cgPortEKG      = 12789
  ,cgPortGUI      = 0
  }
```