

# Cardano.BM - logging, benchmarking and monitoring

Alexander Diemand

Andreas Triantafyllos

November 2018

### **Abstract**

This is a framework that combines logging, benchmarking and monitoring. Complex evaluations of STM or monadic actions can be observed from outside while reading operating system counters before and after, and calculating their differences, thus relating resource usage to such actions. Through interactive configuration, the runtime behaviour of logging or the measurement of resource usage can be altered. Further reduction in logging can be achieved by redirecting log messages to an aggregation function which will output the running statistics with less frequency than the original message.

# Contents

<b>1</b>	<b>Logging, benchmarking and monitoring</b>	<b>3</b>
1.1	Overview	3
1.2	Requirements	3
1.2.1	Observables	3
1.2.2	Traces	4
1.2.3	Aggregation	5
1.2.4	Monitoring	5
1.2.5	Reporting	6
1.2.6	Visualisation	6
1.3	Description	6
1.3.1	Logging with Trace	6
1.3.2	Micro-benchmarks record observables	6
1.3.3	Configuration	8
1.3.4	Information reduction in Aggregation	8
1.3.5	Output selection	8
1.3.6	Monitoring	8
1.4	Examples	8
1.4.1	Simple example showing plain logging	8
1.4.2	Complex example showing logging, aggregation, and observing <i>IO</i> actions	9
1.5	Code listings	16
1.5.1	Cardano.BM.Observer.STM	16
1.5.2	Cardano.BM.Observer.Monad	17
1.5.3	BaseTrace	21
1.5.4	Cardano.BM.Trace	22
1.5.5	Cardano.BM.Setup	27
1.5.6	Cardano.BM.Counters	28
1.5.7	Cardano.BM.Counters.Common	29
1.5.8	Cardano.BM.Counters.Dummy	30
1.5.9	Cardano.BM.Counters.Linux	30
1.5.10	Cardano.BM.Data.Aggregated	37
1.5.11	Cardano.BM.Data.AggregatedKind	43
1.5.12	Cardano.BM.Data.Backend	43
1.5.13	Cardano.BM.Data.BackendKind	44
1.5.14	Cardano.BM.Data.Configuration	44
1.5.15	Cardano.BM.Data.Counter	46
1.5.16	Cardano.BM.Data.LogItem	47
1.5.17	Cardano.BM.Data.Observable	49

1.5.18	Cardano.BM.Data.Output	49
1.5.19	Cardano.BM.Data.Rotation	50
1.5.20	Cardano.BM.Data.Severity	51
1.5.21	Cardano.BM.Data.SubTrace	51
1.5.22	Cardano.BM.Data.Trace	52
1.5.23	Cardano.BM.Configuration	52
1.5.24	Cardano.BM.Configuration.Model	53
1.5.25	Cardano.BM.Configuration.Static	61
1.5.26	Cardano.BM.Configuration.Editor	62
1.5.27	Cardano.BM.Output.Switchboard	66
1.5.28	Cardano.BM.Output.Log	70
1.5.29	Cardano.BM.Output.EKGView	78
1.5.30	Cardano.BM.Output.Aggregation	82
1.5.31	Cardano.BM.Output.Monitoring	87
<b>2</b>	<b>Testing</b>	<b>95</b>
2.1	Test coverage	95
2.2	Test main entry point	95
2.3	Test case generation	97
2.3.1	instance Arbitrary Aggregated	97
2.4	Tests	98
2.4.1	Testing aggregation	98
2.4.2	Cardano.BM.Test.STM	102
2.4.3	Cardano.BM.Test.Trace	102
2.4.4	Testing configuration	114
2.4.5	Rotator	122

# Chapter 1

## Logging, benchmarking and monitoring

### 1.1 Overview

In figure 1.1 we display the relationships among modules in *Cardano.BM*. Central is the **Switchboard** (see `Cardano.BM.Output.Switchboard`) that will redirect incoming log messages to selected backends according the **Configuration** (see `Cardano.BM.Configuration.Model`). The log items are created in the application's context and passed via a hierarchy of **Traces** (see `Cardano.BM.Trace`). Such a hierarchy can be built with the function **subTrace**. The newly added child **Trace** will add its name to the logging context and behave as configured. Among the different kinds of **Traces** implemented are **NoTrace** which suppresses all log items, **FilterTrace** which filters the log items passing through it, and **ObservableTrace** which allows capturing of operating system counters (see `Cardano.BM.Data.SubTrace`). The backend **EKGView** (see `Cardano.BM.Output.EKGView`) displays selected values in a browser. The **Log** backend is based on *katip* and outputs log items in files or the console. The format can be chosen to be textual or JSON representation. And finally, the **Aggregation** backend computes simple statistics over incoming log items (e.g. last, min, max, mean, etc.) (see `Cardano.BM.Data.Aggregated`).

Output selection determines which log items of a named context are routed to which backend. In the case of the **Log** output, this includes a configured output sink (e.g. which file). Items that are aggregated lead to the creation of an output of their current statistics. To prevent a potential infinite loop these aggregation statistics cannot be routed again back into the **Aggregation**.

With *Monitoring* we aim to shortcut the logging-analysis cycle and immediately evaluate monitors on logged values when they become available. In case a monitor is triggered a number of actions can be run: either internal actions that can alter the **Configuration**, or actions that can lead to alerting in external systems.

It is not the intention that this framework should (as part of normal use) record sufficient information so as to make the sequence of events reproducible, i.e. it is not an audit or transaction log.

### 1.2 Requirements

#### 1.2.1 Observables

We can observe the passage of the flow of execution through particular points in the code (really the points at which the graph is reduced). Typically observables would be part of an outcome (which has a start and an end). Where the environment permits these outcomes could also

gather additional environmental context (e.g read system counters, ‘know’ the time). The proposed framework would be able to aggregate, filter such outcome measures so as to calculation things (where appropriate) such as:

- min/max/mean/variance of the resource costs of achieving an outcome
- elapsed wall-clock time
- CPU cycles
- memory allocations, etc
- exponentially weighted moving average of outcomes, events
- min/max/mean/variance of inter-arrival times of demand for service (the arrival pattern)
- measuring offered load against the system (e.g rate/distribution of requests against the wall-let by an exchange, transactions being forwarded between nodes)

### STM evaluation

We treat STM evaluation as a black box and register measurables (counters) before entering, and report the difference at exit together with the result. Logging in an STM will keep a list of log items which at the exit of the evaluation will be passed to the logging subsystem. Since we do not know the exact time an event occurred in the STM action, we annotate the event afterwards with the time interval of the STM action.

### Function evaluation

We treat a function call as a black box and register measurables (counters) before entering, and report the difference at exit together with the result. The function is expected to accept a ‘Trace’ argument which receives the events.

### QuickCheck properties *tentatively*

The function

```
quickCheckResult :: Testable prop => prop -> IO Result
```

will return a *Result* data structure from which we can extract the number of tests performed. Recording the start and end times allows us to derive the time spent for a single test. (although this measurement is wrong as it includes the time spent in QuickCheck setting up the test case (and shrinking?))

#### 1.2.2 Traces

Log items are sent as streams of events to the logging system for processing (aggregation, ..) before output. Functions that need to log events must accept a *Trace* argument. There is no monad related to logging in the monad stack, thus this can work in any monadic environment.

### Trace Context

A Trace maintains a named context stack. A new name can be put onto it, and all subsequent log messages are labeled with this named context. This is also true to all downstream functions which receive the modified Trace. We thus can see the call tree and how the evaluation entered the context where a logging function was called. The context also maintains a mapping from name to Severity: this way a logging function call can early end and not produce a log item when the minimum severity is not reached.

### SubTrace

A Trace is created in *IO* within `setupTrace` with the intent to pass the traced items to a downstream logging framework for outputting to various destinations in different formats. Apart from adding a name to the naming stack we can also alter the behaviour of the Trace. The newly created Trace with a specific function to process the recorded items will forward these to the upstream Trace. This way we can, for example, locally turn on aggregation of observables and only report a summary to the logs.

#### 1.2.3 Aggregation

Log items contain a named context, severity and a payload (message, structured value). Thinking of a relation

$(name, severity) \rightarrow value$

, folding a summarizing function over it outputs

$(name, severity) \rightarrow Summary$

. Depending on the type of *value*, the summary could provide for example:

- $*$  : first, last, count, the time between events (mean, sigma)
- Num : min, max, median, quartiles, mean, sigma, the delta between events (mean, sigma)

Other possible aggregations:

- exponentially weighted moving average
- histograms

#### 1.2.4 Monitoring

- Enable (or disable) measuring events and performance at runtime (e.g. measure how block holding time has changed).
- Send alarms when observables give evidence for abnormalities
- Observe actions in progress, i.e. have started and not yet finished
- Bridge to *Datadog*?

### 1.2.5 Reporting

We might want to buffer events in case an exception is detected. This FIFO queue could then be output to the log for post-factum inspection.

### 1.2.6 Visualisation

#### EKG

<https://hackage.haskell.org/package/ekg>

This library allows live monitor a running instance over HTTP. There is a way we can add our own metrics to it and update them.

#### Log files

The output of observables immediately or aggregated to log files. The format is chosen to be JSON for easier post-processing.

#### Web app

Could combine EKG, log files and parameterization into one GUI.  
(e.g. <https://github.com/HeinrichApfelmus/threepenny-gui>)

## 1.3 Description

### 1.3.1 Logging with **Trace**

#### Setup procedure

#### Hierarchy of **Traces**

### 1.3.2 Micro-benchmarks record observables

Micro-benchmarks are recording observables that measure resource usage of the whole program for a specific time. These measurements are then associated with the subsystem that was observed at that time. Caveat: if the executable under observation runs on a multiprocessor computer where more than one parallel thread executes at the same time, it becomes difficult to associate resource usage to a single function. Even more so, as Haskell's thread do not map directly to operating system threads. So the expressiveness of our approach is only valid statistically when a large number of observables have been captured.

#### Counters

The framework provides access to the following O/S counters (defined in **ObservableInstance**) on *Linux*:

- monotonic clock (see **MonotonicClock**)
- CPU or total time (`/proc/<pid>/stat`) (see **ProcessStats**)
- memory allocation (`/proc/<pid>/statm`) (see **MemoryStats**)



- network bytes received/sent (`/proc/<pid>/net/netstat`) (see [NetStats](#))
- disk input/output (`/proc/<pid>/io`) (see [IOStats](#))

On all platforms, access is provided to the RTS counters (see [GhcRtsStats](#)).

### Implementing micro-benchmarks

In a micro-benchmark we capture operating system counters over an STM evaluation or a function, before and afterwards. Then, we compute the difference between the two and report all three measurements via a *Trace* to the logging system. Here we refer to the example that can be found in [complex example](#).

The capturing of STM actions is defined in [Cardano.BM.Observer.STM](#) and the function `STM.bracketObserveIO` has type:

```
STM.bracketObserveIO trace "observeSTM" (stmAction args)
```

```
bracketObserveIO :: Trace IO -> Text -> STM.STM t -> IO t
```

It accepts a *Trace* to which it logs, adds a name to the context name and enters this with a *SubTrace*, and finally the STM action which will be evaluated. Because this evaluation can be retried, we cannot pass to it a *Trace* to which it could log directly. A variant of this function [bracketObserveLogIO](#) also captures log items in its result, which then are threaded through the *Trace*.

Capturing observables for a function evaluation in *IO*, the type of `bracketObserveIO` (defined in [Cardano.BM.Observer.Monad](#)) is:

```
bracketObserveIO :: Trace IO -> Text -> IO t -> IO t
```

It accepts a *Trace* to which it logs items, adds a name to the context name and enters this with a *SubTrace*, and then the IO action which will be evaluated.

Counters are evaluated before the evaluation and afterwards. We trace these as log items [ObserveOpen](#) and [ObserveClose](#), as well as the difference with type [ObserveDiff](#).

```
bracketObserveIO trace "observeDownload" $ do
  license <- openURI "http://www.gnu.org/licenses/gpl.txt"
  case license of
    Right bs -> logInfo trace $ pack $ BS8.unpack bs
    Left e    -> logError trace $ "failed to download; error: " ++ (show e)
  threadDelay 50000 -- .05 second
  pure ()
```

### Configuration of mu-benchmarks

Observed STM actions or functions enter a new named context with a *SubTrace*. Thus, they need a configuration of the behaviour of this *SubTrace* in the new context. We can define this in the configuration for our example:

```
CM.setSubTrace c "complex.observeDownload" (Just $ ObservableTrace [NetStats,IOStats])
```

This enables the capturing of network and I/O stats from the operating system. Other Observables are implemented in [Cardano.BM.Data.Observable](#).

Captured observables need to be routed to backends. In our example we configure:

```
CM.setBackends c "complex.observeIO" (Just [AggregationBK])
```

to direct observables from named context `complex.observeIO` to the Aggregation backend.

### 1.3.3 Configuration

#### Format

The configuration is parsed from a file in *Yaml* format (see <https://en.wikipedia.org/wiki/YAML>) on startup. In a first parsing step the file is loaded into an internal *Representation*. This structure is then further processed and validated before copied into the runtime *Configuration*.

#### Configuration editor

The configuration editor (figure 1.3) provides a minimalistic GUI accessible through a browser that directly modifies the runtime configuration of the logging system. Most importantly, the global minimum severity filter can be set. This will suppress all log messages that have a severity assigned that is lower than this setting. Moreover, the following behaviours of the logging system can be changed through the GUI:

- *Backends*: relates the named logging context to a *BackendKind*
- *Scribes*: if the backend is *KatipBK*, defines to which outputs the messages are directed (see *ScribeId*)
- *Severities* a local minimum severity filter for just the named context (see *Severity*)
- *SubTrace* entering a new named context can create a new *Trace* with a specific behaviour (see *SubTrace*)
- *Aggregation* if the backend is *AggregationBK*, defines which aggregation method to use (see *AggregatedKind*)

### 1.3.4 Information reduction in *Aggregation*

#### Statistics

#### Configuration

### 1.3.5 Output selection

#### Configuration

### 1.3.6 Monitoring

#### Configuration

#### Evaluation of monitors

#### Actions fired

## 1.4 Examples

### 1.4.1 Simple example showing plain logging

```
{-# LANGUAGE OverloadedStrings #-}
module Main
```

```

    (main)
    where
import Control.Concurrent (threadDelay)
import Cardano.BM.Configuration.Static (defaultConfigStdout)
import Cardano.BM.Setup (setupTrace)
import Cardano.BM.Trace (logDebug, logError, logInfo, logNotice,
                        logWarning)

main :: IO ()
main = do
    c ← defaultConfigStdout
    tr ← setupTrace (Right c) "simple"

    logDebug tr "this is a debug message"
    logInfo tr "this is an information."
    logNotice tr "this is a notice!"
    logWarning tr "this is a warning!"
    logError tr "this is an error!"

    threadDelay 80000
    return ()

```

### 1.4.2 Complex example showing logging, aggregation, and observing IO actions

#### Module header and import directives

```

{-# LANGUAGE CPP #-}
{-# LANGUAGE OverloadedStrings #-}
# if defined (linux_HOST_OS)
# define LINUX
# endif

{-define the parallel procedures that create messages -}
# define RUN_ProcMessageOutput
# define RUN_ProcObserveIO
# define RUN_ProcObseverSTM
# define RUN_ProcObseveDownload
# define RUN_ProcRandom

module Main
    (main)
    where

import Control.Concurrent (threadDelay)
import qualified Control.Concurrent.Async as Async
import Control.Monad (forM, forM_)
import GHC.Conc.Sync (STM, TVar, atomically, newTVar, readTVar, writeTVar)
import Data.Text (pack)
# ifdef LINUX
import qualified Data.ByteString.Char8 as BS8
import Network.Download (openURI)

```

```

# endif
import System.Random
# ifdef ENABLE_GUI
import qualified Cardano.BM.Configuration.Editor as CME
# endif
import qualified Cardano.BM.Configuration.Model as CM
import Cardano.BM.Data.Aggregated (Measurable (..))
import Cardano.BM.Data.AggregatedKind
import Cardano.BM.Data.BackendKind
import Cardano.BM.Data.LogItem
import Cardano.BM.Data.Observable
import Cardano.BM.Data.Output
import Cardano.BM.Data.Rotation
import Cardano.BM.Data.Severity
import Cardano.BM.Data.SubTrace
import Cardano.BM.Observer.Monad (bracketObserveIO)
import qualified Cardano.BM.Observer.STM as STM
import Cardano.BM.Setup
import Cardano.BM.Trace

```

### Define configuration

Selected values can be viewed in EKG on <http://localhost:12789>.

The configuration editor listens on <http://localhost:13789>.

```

config :: IO CM.Configuration
config = do
  c ← CM.empty
  CM.setMinSeverity c Debug
  CM.setSetupBackends c [KatipBK
    # ifdef ENABLE_AGGREGATION
      ,AggregationBK
    # endif
    # ifdef ENABLE_EKG
      ,EKGViewBK
    # endif
  ]
  CM.setDefaultBackends c [KatipBK]
  CM.setSetupScribes c [ScribeDefinition {
    scName = "stdout"
    ,scKind = StdoutSK
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
    ,ScribeDefinition {
    scName = "logs/out.odd.json"
    ,scKind = FileJsonSK
  }
  ]

```

```

    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
  ,ScribeDefinition {
    scName = "logs/out.even.json"
    ,scKind = FileJsonSK
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
  ,ScribeDefinition {
    scName = "logs/downloading.json"
    ,scKind = FileJsonSK
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
  ,ScribeDefinition {
    scName = "logs/out.txt"
    ,scKind = FileTextSK
    ,scPrivacy = ScPublic
    ,scRotation = Just $ RotationParameters
      {rpLogLimitBytes = 5000 -- 5kB
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = 3
      }
  }
]
CM.setDefaultScribes c ["StdoutSK::stdout"]
CM.setScribes c "complex.random" (Just ["StdoutSK::stdout", "FileTextSK::logs/out.txt"])
CM.setScribes c "#aggregated.complex.random" (Just ["StdoutSK::stdout"])
forM_ [(1 :: Int)..10] $ \x →
  if odd x
  then
    CM.setScribes c ("#aggregation.complex.observeSTM." <> (pack $ show x)) $ Just ["FileJsonSK::logs/out.txt"]
  else
    CM.setScribes c ("#aggregation.complex.observeSTM." <> (pack $ show x)) $ Just ["FileJsonSK::logs/out.txt"]
# ifdef LINUX
CM.setSubTrace c "complex.observeDownload" (Just $ ObservableTrace [IOStats, NetStats])
CM.setBackends c "complex.observeDownload" (Just [KatipBK])
CM.setScribes c "complex.observeDownload" (Just ["StdoutSK::stdout", "FileJsonSK::logs/downloading.json"])
# endif
CM.setSubTrace c "complex.random" (Just $ TeeTrace "ewma")
CM.setSubTrace c "#ekgview"
  (Just $ FilterTrace [(Drop (StartsWith "#ekgview.#aggregation.complex.random"),
    Unhide [(EndsWith ".count"),
      (EndsWith ".avg"),
      (EndsWith ".mean")]),
    (Drop (StartsWith "#ekgview.#aggregation.complex.observeIO"),

```

```

    Unhide [(Contains "diff.RTS.cpuNs.timed.")],
    (Drop (StartsWith "#ekgview.#aggregation.complex.observeSTM"),
    Unhide [(Contains "diff.RTS.gcNum.timed.")],
    (Drop (StartsWith "#ekgview.#aggregation.complex.message"),
    Unhide [(Contains ".timed.m")])
    ])
CM.setSubTrace c "complex.observeIO" (Just $ ObservableTrace [GhcRtsStats, MemoryStats])
forM_ [(1 :: Int)..10] $ \x →
    CM.setSubTrace
    c
    ("complex.observeSTM." <> (pack $ show x))
    (Just $ ObservableTrace [GhcRtsStats, MemoryStats])
# ifdef ENABLE_AGGREGATION
    CM.setBackends c "complex.message" (Just [AggregationBK, KatipBK])
    CM.setBackends c "complex.random" (Just [AggregationBK, KatipBK])
    CM.setBackends c "complex.random.ewma" (Just [AggregationBK])
    CM.setBackends c "complex.observeIO" (Just [AggregationBK])
# endif
    forM_ [(1 :: Int)..10] $ \x → do
# ifdef ENABLE_AGGREGATION
        CM.setBackends c
            ("complex.observeSTM." <> (pack $ show x))
            (Just [AggregationBK])
# endif
        CM.setBackends c
            ("#aggregation.complex.observeSTM." <> (pack $ show x))
            (Just [KatipBK])

    CM.setAggregatedKind c "complex.random.rr" (Just StatsAK)
    CM.setAggregatedKind c "complex.random.ewma.rr" (Just (EwmaAK 0.42))
# ifdef ENABLE_EKG
    CM.setBackends c "#aggregation.complex.message" (Just [EKGViewBK])
    CM.setBackends c "#aggregation.complex.observeIO" (Just [EKGViewBK])
    CM.setBackends c "#aggregation.complex.random" (Just [EKGViewBK])
    CM.setBackends c "#aggregation.complex.random.ewma" (Just [EKGViewBK])
    CM.setEKGport c 12789
# endif
    CM.setGUIport c 13789
    return c

```

### Thread that outputs a random number to a **Trace**

```

randomThr :: Trace IO → IO (Async.Async ())
randomThr trace = do
    logInfo trace "starting random generator"
    trace' ← subTrace "random" trace
    proc ← Async.async (loop trace')

```

```

return proc
where
  loop tr = do
    threadDelay 500000 -- 0.5 second
    num ← randomRIO (42 - 42, 42 + 42) :: IO Double
    lo ← LogObject < $ > (mkLOMeta Debug) < * > pure (LogValue "rr" (PureD num))
    traceConditionally tr lo
  loop tr

```

### Thread that observes an IO action

```

observeIO :: Trace IO → IO (Async.Async ())
observeIO trace = do
  logInfo trace "starting observer"
  proc ← Async.async (loop trace)
  return proc
where
  loop tr = do
    threadDelay 5000000 -- 5 seconds
    _ ← bracketObserveIO tr Debug "observeIO" $ do
      num ← randomRIO (100000, 200000) :: IO Int
      ls ← return $ reverse $ init $ reverse $ 42 : [1..num]
      pure $ const ls ()
    loop tr

```

### Threads that observe STM actions on the same TVar

```

observeSTM :: Trace IO → IO [Async.Async ()]
observeSTM trace = do
  logInfo trace "starting STM observer"
  tvar ← atomically $ newTVar ([1..1000] :: [Int])
  -- spawn 10 threads
  proc ← forM ([1 :: Int]..10) $ \x → Async.async (loop trace tvar (pack $ show x))
  return proc
where
  loop tr tvarlist name = do
    threadDelay 10000000 -- 10 seconds
    STM.bracketObserveIO tr Debug ("observeSTM." <> name) (stmAction tvarlist)
    loop tr tvarlist name
stmAction :: TVar [Int] → STM ()
stmAction tvarlist = do
  list ← readTVar tvarlist
  writeTVar tvarlist $ reverse $ init $ reverse $ list
  pure ()

```

**Thread that observes an IO action which downloads a text in order to observe the I/O statistics**

```
# ifdef LINUX
observeDownload :: Trace IO → IO (Async.Async ())
observeDownload trace = do
  proc ← Async.async (loop trace)
  return proc
where
  loop tr = do
    threadDelay 1000000 -- 1 second
    tr' ← appendName "observeDownload" tr
    bracketObserveIO tr' Debug "" $ do
      license ← openURI "http://www.gnu.org/licenses/gpl.txt"
      case license of
        Right bs → logNotice tr' $ pack $ BS8.unpack bs
        Left _ → return ()
      threadDelay 50000 -- .05 second
      pure ()
    loop tr
# endif
```

**Thread that periodically outputs a message**

```
msgThr :: Trace IO → IO (Async.Async ())
msgThr trace = do
  logInfo trace "start messaging .."
  trace' ← subTrace "message" trace
  Async.async (loop trace')
where
  loop tr = do
    threadDelay 3000000 -- 3 seconds
    logNotice tr "N O T I F I C A T I O N ! ! !"
    logDebug tr "a detailed debug message."
    logError tr "Boooooommm .."
    loop tr
```

**Main entry point**

```
main :: IO ()
main = do
  -- create configuration
  c ← config
  # ifdef ENABLE_GUI
  -- start configuration editor
  CME.startup c
```



```

# endif
-- create initial top-level Trace
tr ← setupTrace (Right c) "complex"
logNotice tr "starting program; hit CTRL-C to terminate"
logInfo tr "watch its progress on http://localhost:12789"
# ifdef RUN_ProcRandom
    {-start thread sending unbounded sequence of random numbers to a trace which aggregates them into a
    procRandom ← randomThr tr
# endif
# ifdef RUN_ProcObserveIO
    -- start thread endlessly reversing lists of random length
    procObsvIO ← observeIO tr
# endif
# ifdef RUN_ProcObseverSTM
    -- start threads endlessly observing STM actions operating on the same TVar
    procObsvSTMs ← observeSTM tr
# endif
# ifdef LINUX
# ifdef RUN_ProcObseveDownload
    -- start thread endlessly which downloads sth in order to check the I/O usage
    procObsvDownload ← observeDownload tr
# endif
# endif
# ifdef RUN_ProcMessageOutput
    -- start a thread to output a text messages every n seconds
    procMsg ← msgThr tr
    -- wait for message thread to finish, ignoring any exception
    _ ← Async.waitCatch procMsg
# endif
# ifdef LINUX
# ifdef RUN_ProcObseveDownload
    -- wait for download thread to finish, ignoring any exception
    _ ← Async.waitCatch procObsvDownload
# endif
# endif
# ifdef RUN_ProcObseverSTM
    -- wait for observer thread to finish, ignoring any exception
    _ ← forM procObsvSTMs Async.waitCatch
# endif
# ifdef RUN_ProcObserveIO
    -- wait for observer thread to finish, ignoring any exception
    _ ← Async.waitCatch procObsvIO
# endif
# ifdef RUN_ProcRandom
    -- wait for random thread to finish, ignoring any exception
    _ ← Async.waitCatch procRandom

```

```
#endif
return ()
```

## 1.5 Code listings

### 1.5.1 Cardano.BM.Observer.STM

```
stmWithLog :: STM.STM (t,[LogObject]) → STM.STM (t,[LogObject])
stmWithLog action = action
```

#### Observe STM action in a named context

With given name, create a **SubTrace** according to **Configuration** and run the passed STM action on it.

```
bracketObserveIO :: Trace IO → Severity → Text → STM.STM t → IO t
bracketObserveIO logTrace0 severity name action = do
  logTrace ← subTrace name logTrace0
  let subtrace = typeofTrace logTrace
  bracketObserveIO' subtrace severity logTrace action
where
  bracketObserveIO' :: SubTrace → Severity → Trace IO → STM.STM t → IO t
  bracketObserveIO' NoTrace _ _ act =
    STM.atomically act
  bracketObserveIO' subtrace sev logTrace act = do
    mCountersid ← observeOpen subtrace sev logTrace
    -- run action; if an exception is caught, then it will be logged and rethrown.
    t ← (STM.atomically act) 'catch' (λ(e :: SomeException) → (logError logTrace (pack (show e)) >> throwM e))
    case mCountersid of
      Left openException →
        -- since observeOpen faced an exception there is no reason to call observeClose
        -- however the result of the action is returned
        logNotice logTrace ("ObserveOpen: " <> pack (show openException))
      Right countersid → do
        res ← observeClose subtrace sev logTrace countersid []
        case res of
          Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
          _ → pure ()
    pure t
```

#### Observe STM action in a named context and output captured log items

The STM action might output messages, which after "success" will be forwarded to the logging trace. Otherwise, this function behaves the same as **Observe STM action in a named context**.

```

bracketObserveLogIO :: Trace IO → Severity → Text → STM.STM (t, [LogObject]) → IO t
bracketObserveLogIO logTrace0 severity name action = do
  logTrace ← subTrace name logTrace0
  let subtrace = typeofTrace logTrace
  bracketObserveLogIO' subtrace severity logTrace action
where
  bracketObserveLogIO' :: SubTrace → Severity → Trace IO → STM.STM (t, [LogObject]) → IO t
  bracketObserveLogIO' NoTrace _ _ act = do
    (t, _) ← STM.atomically $ stmWithLog act
    pure t
  bracketObserveLogIO' subtrace sev logTrace act = do
    mCountersid ← observeOpen subtrace sev logTrace
    -- run action, return result and log items; if an exception is
    -- caught, then it will be logged and rethrown.
    (t, as) ← (STM.atomically $ stmWithLog act) 'catch'
      (λ(e :: SomeException) → (logError logTrace (pack (show e)) >> throwM e))
    case mCountersid of
      Left openException →
        -- since observeOpen faced an exception there is no reason to call observeClose
        -- however the result of the action is returned
        logNotice logTrace ("ObserveOpen: " <> pack (show openException))
      Right countersid → do
        res ← observeClose subtrace sev logTrace countersid as
        case res of
          Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
          _ → pure ()
    pure t

```

### 1.5.2 Cardano.BM.Observer.Monadic

#### Monadic.bracketObserverIO

Observes an IO action and adds a name to the logger name of the passed in **Trace**. An empty **Text** leaves the logger name untouched.

Microbenchmarking steps:

1. Create a *trace* which will have been configured to observe things besides logging.

```

import qualified Cardano.BM.Configuration.Model as CM
ooo
c ← config
trace@(ctx, _) ← setupTrace (Right c) "demo-playground"
where
  config :: IO CM.Configuration
  config = do
    c ← CM.empty
    CM.setMinSeverity c Debug

```

```

CM.setSetupBackends c [KatipBK, AggregationBK]
CM.setDefaultBackends c [KatipBK, AggregationBK]
CM.setSetupScribes c [ScribeDefinition {
  scName = "stdout"
  ,scKind = StdoutSK
  ,scRotation = Nothing
}]
CM.setDefaultScribes c ["StdoutSK: stdout"]
return c

```

2. *c* is the **Configuration** of *trace*. In order to enable the collection and processing of measurements (min, max, mean, std-dev) *AggregationBK* is needed.

```
CM.setDefaultBackends c [KatipBK, AggregationBK]
```

in a configuration file (YAML) means

```

defaultBackends:
- KatipBK
- AggregationBK

```

3. Set the measurements that you want to take by changing the configuration of the *trace* using **setSubTrace**, in order to declare the namespace where we want to enable the particular measurements and the list with the kind of measurements.

```

CM.setSubTrace
  (configuration ctx)
  "demo-playground.submit-tx"
  (Just $ ObservableTrace observablesSet)
where
  observablesSet = [MonotonicClock, MemoryStats]

```

4. Find an action to measure. e.g.:

```
runProtocolWithPipe x hdl proto 'catch' (λProtocolStopped → return ())
```

and use **bracketObserveIO**. e.g.:

```

bracketObserveIO trace "submit-tx" $
  runProtocolWithPipe x hdl proto 'catch' (λProtocolStopped → return ())

```

---

```

bracketObserveIO :: Trace IO → Severity → Text → IO t → IO t
bracketObserveIO logTrace0 severity name action = do
  logTrace ← subTrace name logTrace0
  bracketObserveIO' (typeofTrace logTrace) severity logTrace action
where
  bracketObserveIO' :: SubTrace → Severity → Trace IO → IO t → IO t

```

```

bracketObserveIO' NoTrace _ _ act = act
bracketObserveIO' subtrace sev logTrace act = do
  mCountersid ← observeOpen subtrace sev logTrace
  -- run action; if an exception is caught will be logged and rethrown.
  t ← act 'catch' (λ(e :: SomeException) → (logError logTrace (pack (show e)) >> throwM e))
  case mCountersid of
    Left openException →
      -- since observeOpen faced an exception there is no reason to call observeClose
      -- however the result of the action is returned
      logNotice logTrace ("ObserveOpen: " <> pack (show openException))
    Right countersid → do
      res ← observeClose subtrace sev logTrace countersid [ ]
      case res of
        Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
        _ → pure ()
  pure t

```

### Monadic.bracketObserverM

Observes a *MonadIO*  $m \Rightarrow m$  action and adds a name to the logger name of the passed in **Trace**. An empty *Text* leaves the logger name untouched.

```

bracketObserveM :: (MonadCatch m, MonadIO m) => Trace IO → Severity → Text → m t → m t
bracketObserveM logTrace0 severity name action = do
  logTrace ← liftIO $ subTrace name logTrace0
  bracketObserveM' (typeofTrace logTrace) severity logTrace action
where
  bracketObserveM' :: (MonadCatch m, MonadIO m) => SubTrace → Severity → Trace IO → m t → m t
  bracketObserveM' NoTrace _ _ act = act
  bracketObserveM' subtrace sev logTrace act = do
    mCountersid ← liftIO $ observeOpen subtrace sev logTrace
    -- run action; if an exception is caught will be logged and rethrown.
    t ← act 'catch'
      (λ(e :: SomeException) → (liftIO (logError logTrace (pack (show e)) >> throwM e)))
    case mCountersid of
      Left openException →
        -- since observeOpen faced an exception there is no reason to call observeClose
        -- however the result of the action is returned
        liftIO $ logNotice logTrace ("ObserveOpen: " <> pack (show openException))
      Right countersid → do
        res ← liftIO $ observeClose subtrace sev logTrace countersid [ ]
        case res of
          Left ex → liftIO (logNotice logTrace ("ObserveClose: " <> pack (show ex)))
          _ → pure ()
    pure t

```

**observerOpen**

```

observeOpen :: SubTrace → Severity → Trace IO → IO (Either SomeException CounterState)
observeOpen subtrace severity logTrace = (do
  identifier ← newUnique
  -- take measurement
  counters ← readCounters subtrace
  let state = CounterState identifier counters
  if counters ≡ []
  then return ()
  else do
    -- send opening message to Trace
    traceConditionally logTrace ≪
      LogObject < $ > (mkLOMeta severity) < * > pure (ObserveOpen state)
    return (Right state)) 'catch' (return ◦ Left)

```

**observeClose**

```

observeClose
  :: SubTrace
  → Severity
  → Trace IO
  → CounterState
  → [LogObject]
  → IO (Either SomeException ())
observeClose subtrace sev logTrace initState logObjects = (do
  let identifier = csIdentifier initState
      initialCounters = csCounters initState
  -- take measurement
  counters ← readCounters subtrace
  if counters ≡ []
  then return ()
  else do
    mle ← mkLOMeta sev
    -- send closing message to Trace
    traceConditionally logTrace $
      LogObject mle (ObserveClose (CounterState identifier counters))
    -- send diff message to Trace
    traceConditionally logTrace $
      LogObject mle (ObserveDiff (CounterState identifier (diffCounters initialCounters counters)))
    -- trace the messages gathered from inside the action
    forM_ logObjects $ traceConditionally logTrace
    return (Right ())) 'catch' (return ◦ Left)

```

### 1.5.3 BaseTrace

#### Contravariant

A covariant is a functor:  $F A \rightarrow F B$

A contravariant is a functor:  $F B \rightarrow F A$

`Op a b` implements the inverse to 'arrow' "`getOp :: b → a`", which when applied to a **BaseTrace** of type "`Op (m ()) s`", yields "`s → m ()`". In our case, `Op` accepts an action in a monad `m` with input type **LogNamed LogObject** (see 'Trace').

```
newtype BaseTrace m s = BaseTrace {runTrace :: Op (m ()) s}
```

#### contramap

A covariant functor defines the function "`fmap :: (a → b) → f a → f b`". In case of a contravariant functor, it is the dual function "`contramap :: (a → b) → f b → f a`" which is defined.

In the following instance, `runTrace` extracts type "`Op (m ()) s`" to which `contramap` applies `f`, thus "`f s → m ()`". The constructor **BaseTrace** restores "`Op (m ()) (f s)`".

```
instance Contravariant (BaseTrace m) where
  contramap f = BaseTrace ∘ contramap f ∘ runTrace
```

#### traceWith

Accepts a **Trace** and some payload `s`. First it gets the contravariant from the **Trace** as type "`Op (m ()) s`" and, after "`getOp :: b → a`" which translates to "`s → m ()`", calls the action on the **LogNamed LogObject**.

```
traceWith :: BaseTrace m s → s → m ()
traceWith = getOp ∘ runTrace
```

#### natTrace

Natural transformation from monad `m` to monad `n`.

```
natTrace :: (forall x ∘ m x → n x) → BaseTrace m s → BaseTrace n s
natTrace nat (BaseTrace (Op tr)) = BaseTrace $ Op $ nat ∘ tr
```

#### noTrace

A **Trace** that discards all inputs.

```
noTrace :: Applicative m ⇒ BaseTrace m a
noTrace = BaseTrace $ Op $ const (pure ())
```

### 1.5.4 Cardano.BM.Trace

#### Utilities

Natural transformation from monad  $m$  to monad  $n$ .

```
natTrace :: (forall  $x \circ m \ x \rightarrow n \ x$ )  $\rightarrow$  Trace  $m \rightarrow$  Trace  $n$ 
natTrace nat (ctx, trace) = (ctx, BaseTrace.natTrace nat trace)
```

Access type of **Trace**.

```
typeofTrace :: Trace  $m \rightarrow$  SubTrace
typeofTrace (ctx, _) = tracetype ctx
```

Update type of **Trace**.

```
updateTracetype :: SubTrace  $\rightarrow$  Trace  $m \rightarrow$  Trace  $m$ 
updateTracetype subtr (ctx, tr) = (ctx { tracetype = subtr }, tr)
```

#### Enter new named context

The context name is created and checked that its size is below a limit (currently 80 chars). The minimum severity that a log message must be labelled with is looked up in the configuration and recalculated.

```
appendName :: MonadIO  $m \Rightarrow$  LoggerName  $\rightarrow$  Trace  $m \rightarrow m$  (Trace  $m$ )
appendName name (ctx, trace) = do
  let prevLoggerName = loggerName ctx
      prevMinSeverity = minSeverity ctx
      newLoggerName = appendWithDot prevLoggerName name
      globMinSeverity  $\leftarrow$  liftIO $ Config.minSeverity (configuration ctx)
      namedSeverity  $\leftarrow$  liftIO $ Config.inspectSeverity (configuration ctx) newLoggerName
  case namedSeverity of
    Nothing  $\rightarrow$  return (ctx { loggerName = newLoggerName }, trace)
    Just sev  $\rightarrow$  return (ctx { loggerName = newLoggerName
      , minSeverity = max (max sev prevMinSeverity) globMinSeverity }
      , trace)

appendWithDot :: LoggerName  $\rightarrow$  LoggerName  $\rightarrow$  LoggerName
appendWithDot "" newName = T.take 80 newName
appendWithDot xs "" = xs
appendWithDot xs newName = T.take 80 $ xs <> "." <> newName
```

#### Change named context

The context name is created and checked that its size is below a limit (currently 80 chars). The minimum severity that a log message must be labelled with is looked up in the configuration and recalculated.

```
modifyName :: MonadIO  $m \Rightarrow$  LoggerName  $\rightarrow$  Trace  $m \rightarrow m$  (Trace  $m$ )
modifyName name (ctx, trace) = do
```



```

let prevMinSeverity = minSeverity ctx
globMinSeverity ← liftIO $ Config.minSeverity (configuration ctx)
namedSeverity ← liftIO $ Config.inspectSeverity (configuration ctx) name
case namedSeverity of
  Nothing → return (ctx {loggerName = name}, trace)
  Just sev → return (ctx {loggerName = name
    , minSeverity = max (max sev prevMinSeverity) globMinSeverity}
    , trace)

```

### Contramap a trace and produce the naming context

```

named :: BaseTrace.BaseTrace m (LogNamed i) → LoggerName → BaseTrace.BaseTrace m i
named trace name = contramap (LogNamed name) trace

```

### Trace a **LogObject** through

```

traceNamedObject
  :: MonadIO m
  ⇒ Trace m
  → LogObject
  → m ()
traceNamedObject trace@(ctx, logTrace) lo@(LogObject _ lc) = do
  let lname = loggerName ctx
  doOutput ← case (typeofTrace trace) of
    FilterTrace filters →
      case lc of
        LogValue lname _ →
          return $ evalFilters filters (lname <> " ." <> lname)
        _ →
          return $ evalFilters filters lname
    TeeTrace secName → do
      -- create a newly named copy of the LogObject
      BaseTrace.traceWith (named logTrace (lname <> " ." <> secName)) lo
      return True
    _ → return True
  if doOutput
  then BaseTrace.traceWith (named logTrace lname) lo
  else return ()

```

### Evaluation of **FilterTrace**

A filter consists of a *DropName* and a list of *UnhideNames*. If the context name matches the *DropName* filter, then at least one of the *UnhideNames* must match the name to have the evaluation of the filters return *True*.

```

evalFilters :: [(DropName, UnhideNames)] → LoggerName → Bool
evalFilters fs nm =
  all (λ(no, yes) → if (dropFilter nm no) then (unhideFilter nm yes) else True) fs
where
  dropFilter :: LoggerName → DropName → Bool
  dropFilter name (Drop sel) = {-not -} (matchName name sel)
  unhideFilter :: LoggerName → UnhideNames → Bool
  unhideFilter _ (Unhide [ ]) = False
  unhideFilter name (Unhide us) = any (λsel → matchName name sel) us
  matchName :: LoggerName → NameSelector → Bool
  matchName name (Exact name') = name ≡ name'
  matchName name (StartsWith prefix) = T.isPrefixOf prefix name
  matchName name (EndsWith postfix) = T.isSuffixOf postfix name
  matchName name (Contains name') = T.isInfixOf name' name

```

### Concrete Trace on stdout

This function returns a trace with an action of type "(LogNamed LogObject) → IO ()" which will output a text message as text and all others as JSON encoded representation to the console.

**TODO** remove *locallock*

```

locallock :: MVar ()
locallock = unsafePerformIO $ newMVar ()

stdoutTrace :: TraceNamed IO
stdoutTrace = BaseTrace.BaseTrace $ Op $ λ(LogNamed logname (LogObject _ lc)) →
  withMVar locallock $ \_ →
    case lc of
      (LogMessage logItem) →
        output logname $ liPayload logItem
      obj →
        output logname $ toStrict (encodeToLazyText obj)
where
  output nm msg = TIO.putStrLn $ nm <> " :: " <> msg

```

### Concrete Trace into a TVar

```

traceInTVar :: STM.TVar [a] → BaseTrace.BaseTrace STM.STM a
traceInTVar tvar = BaseTrace.BaseTrace $ Op $ λa → STM.modifyTVar tvar ((:) a)
traceInTVarIO :: STM.TVar [LogObject] → TraceNamed IO
traceInTVarIO tvar = BaseTrace.BaseTrace $ Op $ λln →
  STM.atomically $ STM.modifyTVar tvar ((:) (lnItem ln))
traceNamedInTVarIO :: STM.TVar [LogNamed LogObject] → TraceNamed IO
traceNamedInTVarIO tvar = BaseTrace.BaseTrace $ Op $ λln →
  STM.atomically $ STM.modifyTVar tvar ((:) ln)

```

Check a log item's severity against the **Trace's** minimum severity

do we need three different **minSeverity** defined?

We do a lookup of the global **minSeverity** in the configuration. And, a lookup of the **minSeverity** for the current named context. These values might have changed in the meanwhile.

A third filter is the **minSeverity** defined in the current context.

```

traceConditionally
  :: MonadIO m
  => Trace m
  → LogObject
  → m ()
traceConditionally logTrace@(ctx, _) msg@(LogObject meta _) = do
  globminsev ← liftIO $ Config.minSeverity (configuration ctx)
  globnamesev ← liftIO $ Config.inspectSeverity (configuration ctx) (loggerName ctx)
  let minsev = max (minSeverity ctx) $ max globminsev $ fromMaybe Debug globnamesev
      flag = (severity meta) ≥ minsev
  when flag $ traceNamedObject logTrace msg

```

Enter message into a trace

The function **traceNamedItem** creates a **LogObject** and threads this through the action defined in the **Trace**.

```

traceNamedItem
  :: MonadIO m
  => Trace m
  → LogSelection
  → Severity
  → T.Text
  → m ()
traceNamedItem trace p s m =
  traceConditionally trace ≪≪
    LogObject < $ > liftIO (mkLOMeta s)
    < * > pure (LogMessage LogItem {liSelection = p
    , liPayload = m
    })

```

Logging functions

```

logDebug, logInfo, logNotice, logWarning, logError, logCritical, logAlert, logEmergency
  :: MonadIO m => Trace m → T.Text → m ()
logDebug  logTrace = traceNamedItem logTrace Both Debug
logInfo    logTrace = traceNamedItem logTrace Both Info
logNotice  logTrace = traceNamedItem logTrace Both Notice
logWarning logTrace = traceNamedItem logTrace Both Warning

```

```

logError    logTrace = traceNamedItem logTrace Both Error
logCritical logTrace = traceNamedItem logTrace Both Critical
logAlert    logTrace = traceNamedItem logTrace Both Alert
logEmergency logTrace = traceNamedItem logTrace Both Emergency
logDebugS, logInfoS, logNoticeS, logWarningS, logErrorS, logCriticalS, logAlertS, logEmergencyS
  :: MonadIO m => Trace m -> T.Text -> m ()
logDebugS   logTrace = traceNamedItem logTrace Private Debug
logInfoS    logTrace = traceNamedItem logTrace Private Info
logNoticeS  logTrace = traceNamedItem logTrace Private Notice
logWarningS logTrace = traceNamedItem logTrace Private Warning
logErrorS   logTrace = traceNamedItem logTrace Private Error
logCriticalS logTrace = traceNamedItem logTrace Private Critical
logAlertS   logTrace = traceNamedItem logTrace Private Alert
logEmergencyS logTrace = traceNamedItem logTrace Private Emergency

```

### subTrace

Transforms the input **Trace** according to the **Configuration** using the logger name of the current **Trace** appended with the new name. If the empty **Text** is passed, then the logger name remains untouched.

```

subTrace :: MonadIO m => T.Text -> Trace m -> m (Trace m)
subTrace name tr@(ctx, _) = do
  let newName = appendWithDot (loggerName ctx) name
  subtrace0 <- liftIO $ Config.findSubTrace (configuration ctx) newName
  let subtrace = case subtrace0 of Nothing -> Neutral; Just str -> str
  case subtrace of
    Neutral      -> do
      tr' <- appendName name tr
      return $ updateTracetype subtrace tr'
    UntimedTrace -> do
      tr' <- appendName name tr
      return $ updateTracetype subtrace tr'
    TeeTrace _   -> do
      tr' <- appendName name tr
      return $ updateTracetype subtrace tr'
    FilterTrace _ -> do
      tr' <- appendName name tr
      return $ updateTracetype subtrace tr'
    NoTrace      -> return $ updateTracetype subtrace (ctx, BaseTrace.BaseTrace $ Op $ \_ -> pure ())
    DropOpening  -> return $ updateTracetype subtrace (ctx, BaseTrace.BaseTrace $ Op $
      \ (LogNamed _ lo@(LogObject _ lc)) -> do
        case lc of
          ObserveOpen _ -> return ()
          _ -> traceConditionally tr lo)
    ObservableTrace _ -> do
      tr' <- appendName name tr
      return $ updateTracetype subtrace tr'

```

### 1.5.5 Cardano.BM.Setup

#### setupTrace

Setup a new **Trace** (**Trace**) with either a given **Configuration** (**Configuration.Model**) or a *FilePath* to a configuration file. After all tracing operations have ended; *shutdownTrace* must be called.

```

setupTrace :: MonadIO m => Either FilePath Config.Configuration -> Text -> m (Trace m)
setupTrace (Left cfgFile) name = do
  c <- liftIO $ Config.setup cfgFile
  fst < $ > setupTrace_c name
setupTrace (Right c) name = fst < $ > setupTrace_c name
setupTrace_c :: MonadIO m => Config.Configuration -> Text -> m (Trace m, Switchboard.Switchboard)
setupTrace_c name = do
  sb <- liftIO $ Switchboard.realize c
  sev <- liftIO $ Config.minSeverity c
  ctx <- liftIO $ newContext " " c sev
  tr <- subTrace name $ natTrace liftIO (ctx, Switchboard.mainTrace sb)
  return (tr, sb)

```

#### shutdown

Shut down the Switchboard and all the **Traces** related to it.

```

shutdown :: Switchboard.Switchboard -> IO ()
shutdown = Switchboard.unrealize

```

#### withTrace

Setup a **Trace** from **Configuration** and pass it to the action. At the end, shutdown all the components and close the trace.

```

withTrace :: (MonadIO m, MonadMask m) => Config.Configuration -> Text -> (Trace m -> m t) -> m t
withTrace cfg name action =
  bracket
    (setupTrace_c cfg name)          -- acquire
    (\(_, sb) -> liftIO $ shutdown sb) -- release
    (\(tr, _) -> action tr)          -- action

```

#### newContext

```

newContext :: LoggerName
  -> Config.Configuration
  -> Severity
  -> IO TraceContext
newContext name cfg sev = do
  return $ TraceContext {

```

```

loggerName = name
,configuration = cfg
,minSeverity = sev
,traceType = Neutral
}

```

### 1.5.6 Cardano.BM.Counters

Here the platform is chosen on which we compile this program.

Currently, we mainly support *Linux* with its 'proc' filesystem.

```

{-# LANGUAGE CPP #-}
# if defined (linux_HOST_OS)
# define LINUX
# endif
module Cardano.BM.Counters
(
    Platform.readCounters
    ,diffTimeObserved
    ,getMonoClock
) where
# ifdef LINUX
import qualified Cardano.BM.Counters.Linux as Platform
# else
import qualified Cardano.BM.Counters.Dummy as Platform
# endif
import Cardano.BM.Counters.Common (getMonoClock)
import Cardano.BM.Data.Aggregated (Measurable (..))
import Cardano.BM.Data.Counter

```

#### Calculate difference between clocks

```

diffTimeObserved :: CounterState → CounterState → Measurable
diffTimeObserved (CounterState id0 startCounters) (CounterState id1 endCounters) =
    let
        startTime = getMonotonicTime startCounters
        endTime = getMonotonicTime endCounters
    in
        if (id0 == id1)
        then endTime - startTime
        else error "these clocks are not from the same experiment"
    where
        getMonotonicTime counters = case (filter isMonotonicClockCounter counters) of
            [(Counter MonotonicClockTime _ mus)] → mus
            _ → error "A time measurement is missing!"

```

```
isMonotonicClockCounter :: Counter → Bool
isMonotonicClockCounter = (MonotonicClockTime ≡) ∘ cType
```

### 1.5.7 Cardano.BM.Counters.Common

Common functions that serve *readCounters* on all platforms.

```
nominalTimeToMicroseconds :: Word64 → Microsecond
nominalTimeToMicroseconds = fromMicroseconds ∘ toInteger ∘ ('div' 1000)
```

#### Read monotonic clock

```
getMonoClock :: IO [Counter]
getMonoClock = do
  t ← getMonotonicTimeNSec
  return [Counter MonotonicClockTime "monoclock" $ Microseconds (t `div` 1000)]
```

#### Read GHC RTS statistics

Read counters from GHC's RTS (runtime system). The values returned are as per the last GC (garbage collection) run.

```
readRTSStats :: IO [Counter]
readRTSStats = do
  iscollected ← GhcStats.getRTSStatsEnabled
  if iscollected
    then ghcstats
    else return []
  where
    ghcstats :: IO [Counter]
    ghcstats = do
      -- need to run GC?
      rts ← GhcStats.getRTSStats
      let getrts = ghcval rts
      return [getrts (Bytes ∘ fromIntegral ∘ GhcStats.allocated_bytes, "bytesAllocated")
        , getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_live_bytes, "liveBytes")
        , getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_large_objects_bytes, "largeBytes")
        , getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_compact_bytes, "compactBytes")
        , getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_slop_bytes, "slopBytes")
        , getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_mem_in_use_bytes, "usedMemBytes")
        , getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.gc_cpu_ns, "gcCpuNs")
        , getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.gc_elapsed_ns, "gcElapsedNs")
        , getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.cpu_ns, "cpuNs")
        , getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.elapsed_ns, "elapsedNs")
        , getrts (PureI ∘ toInteger ∘ GhcStats.gcs, "gcNum")
        , getrts (PureI ∘ toInteger ∘ GhcStats.major_gcs, "gcMajorNum")]
```

```

]
ghcval :: GhcStats.RTSStats → ((GhcStats.RTSStats → Measurable), Text) → Counter
ghcval s (f, n) = Counter RTSStats n $ (f s)

```

### 1.5.8 Cardano.BM.Counters.Dummy

This is a dummy definition of `readCounters` on platforms that do not support the 'proc' filesystem from which we would read the counters.

The only supported measurements are monotonic clock time and RTS statistics for now.

```

readCounters :: SubTrace → IO [Counter]
readCounters NoTrace      = return []
readCounters Neutral      = return []
readCounters (TeeTrace _) = return []
readCounters (FilterTrace _) = return []
readCounters UntimedTrace = return []
readCounters DropOpening  = return []
readCounters (ObservableTrace tts) = foldrM (\(sel, fun) a →
  if any (≡ sel) tts
  then (fun >>= λxs → return $ a ++ xs)
  else return a) [] selectors
where
  selectors = [(MonotonicClock, getMonoClock)
    , (GhcRtsStats, readRTSStats)
    ]

```

### 1.5.9 Cardano.BM.Counters.Linux

we have to expand the `readMemStats` function to read full data from `proc`

```

readCounters :: SubTrace → IO [Counter]
readCounters NoTrace      = return []
readCounters Neutral      = return []
readCounters (TeeTrace _) = return []
readCounters (FilterTrace _) = return []
readCounters UntimedTrace = return []
readCounters DropOpening  = return []
readCounters (ObservableTrace tts) = do
  pid ← getProcessID
  foldrM (\(sel, fun) a →
    if any (≡ sel) tts
    then (fun >>= λxs → return $ a ++ xs)
    else return a) [] (selectors pid)
where
  selectors pid = [(MonotonicClock, getMonoClock)
    , (MemoryStats, readProcStatM pid)
    ]

```



```
,(ProcessStats,readProcStats pid)
,(NetStats,readProcNet pid)
,(IOStats,readProcIO pid)
,(GhcRtsStats,readRTSStats)
]
```

```
pathProc :: FilePath
pathProc = "/proc/"
pathProcStat :: ProcessID → FilePath
pathProcStat pid = pathProc </> (show pid) </> "stat"
pathProcStatM :: ProcessID → FilePath
pathProcStatM pid = pathProc </> (show pid) </> "statm"
pathProcIO :: ProcessID → FilePath
pathProcIO pid = pathProc </> (show pid) </> "io"
pathProcNet :: ProcessID → FilePath
pathProcNet pid = pathProc </> (show pid) </> "net" </> "netstat"
```

### Reading from a file in /proc/<pid>

```
readProcList :: FilePath → IO [Integer]
readProcList fp = do
  cs ← readFile fp
  return $ map (λs → maybe 0 id $ (readMaybe s :: Maybe Integer)) (words cs)
```

### readProcStatM - /proc/<pid>/statm

```
/proc/[pid]/statm
Provides information about memory usage, measured in pages. The columns are:
size      (1) total program size
           (same as VmSize in /proc/[pid]/status)
resident  (2) resident set size
           (same as VmRSS in /proc/[pid]/status)
shared    (3) number of resident shared pages (i.e., backed by a file)
           (same as RssFile+RssShmem in /proc/[pid]/status)
text      (4) text (code)
lib       (5) library (unused since Linux 2.6; always 0)
data      (6) data + stack
dt        (7) dirty pages (unused since Linux 2.6; always 0)
```

```
readProcStatM :: ProcessID → IO [Counter]
readProcStatM pid = do
  ps0 ← readProcList (pathProcStatM pid)
  let ps = zip colnames ps0
  psUseful = filter ((("unused" ≠) ∘ fst) ps) ps
  return $ map (λ(n,i) → Counter MemoryCounter n (PureI i)) psUseful
where
  colnames :: [Text]
  colnames = ["size", "resident", "shared", "text", "unused", "data", "unused"]
```

**readProcStats - //proc//<pid >//stat**

/proc/[pid]/stat

Status information about the process. This is used by ps(1). It is defined in the kernel source file fs/proc/array.c.

The fields, in order, with their proper scanf(3) format specifiers, are listed below. Whether or not certain of these fields display valid information is governed by a ptrace access mode PTRACE\_MODE\_READ\_FSCREDS | PTRACE\_MODE\_NOAUDIT check (refer to ptrace(2)). If the check denies access, then the field value is displayed as 0. The affected fields are indicated with the marking [PT].

- (1) pid %d  
The process ID.
- (2) comm %s  
The filename of the executable, in parentheses. This is visible whether or not the executable is swapped out.
- (3) state %c  
One of the following characters, indicating process state:
  - R Running
  - S Sleeping in an interruptible wait
  - D Waiting in uninterruptible disk sleep
  - Z Zombie
  - T Stopped (on a signal) or (before Linux 2.6.33) trace stopped
  - t Tracing stop (Linux 2.6.33 onward)
  - W Paging (only before Linux 2.6.0)
  - X Dead (from Linux 2.6.0 onward)
  - x Dead (Linux 2.6.33 to 3.13 only)
  - K Wakekill (Linux 2.6.33 to 3.13 only)
  - W Waking (Linux 2.6.33 to 3.13 only)
  - P Parked (Linux 3.9 to 3.13 only)
- (4) ppid %d  
The PID of the parent of this process.
- (5) pgrp %d  
The process group ID of the process.
- (6) session %d  
The session ID of the process.
- (7) tty\_nr %d  
The controlling terminal of the process. (The minor device number is contained in the combination of bits 31 to 20 and 7 to 0; the major device number is in bits 15 to 8.)
- (8) tpgid %d  
The ID of the foreground process group of the controlling terminal of the process.
- (9) flags %u  
The kernel flags word of the process. For bit meanings, see the PF\_\* defines in the Linux kernel source file include/linux/sched.h. Details depend on the kernel version.  
  
The format for this field was %lu before Linux 2.6.
- (10) minflt %lu

- The number of minor faults the process has made which have not required loading a memory page from disk.
- (11) `cminflt` %lu  
The number of minor faults that the process's waited-for children have made.
- (12) `majflt` %lu  
The number of major faults the process has made which have required loading a memory page from disk.
- (13) `cmajflt` %lu  
The number of major faults that the process's waited-for children have made.
- (14) `utime` %lu  
Amount of time that this process has been scheduled in user mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`). This includes guest time, `guest_time` (time spent running a virtual CPU, see below), so that applications that are not aware of the guest time field do not lose that time from their calculations.
- (15) `stime` %lu  
Amount of time that this process has been scheduled in kernel mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- (16) `cutime` %ld  
Amount of time that this process's waited-for children have been scheduled in user mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`). (See also `times(2)`.) This includes guest time, `cguest_time` (time spent running a virtual CPU, see below).
- (17) `cstime` %ld  
Amount of time that this process's waited-for children have been scheduled in kernel mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- (18) `priority` %ld  
(Explanation for Linux 2.6) For processes running a real-time scheduling policy (policy below; see `sched_setscheduler(2)`), this is the negated scheduling priority, minus one; that is, a number in the range -2 to -100, corresponding to real-time priorities 1 to 99. For processes running under a non-real-time scheduling policy, this is the raw nice value (`setpriority(2)`) as represented in the kernel. The kernel stores nice values as numbers in the range 0 (high) to 39 (low), corresponding to the user-visible nice range of -20 to 19.
- (19) `nice` %ld  
The nice value (see `setpriority(2)`), a value in the range 19 (low priority) to -20 (high priority).
- (20) `num_threads` %ld  
Number of threads in this process (since Linux 2.6). Before kernel 2.6, this field was hard coded to 0 as a placeholder for an earlier removed field.
- (21) `itrealvalue` %ld  
The time in jiffies before the next `SIGALRM` is sent to the process due to an interval timer. Since kernel 2.6.17, this field is no longer maintained, and is hard coded as 0.
- (22) `starttime` %llu  
The time the process started after system boot. In kernels before Linux 2.6, this value was expressed in jiffies. Since Linux 2.6, the value is expressed in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).  
  
The format for this field was %lu before Linux 2.6.
- (23) `vsize` %lu  
Virtual memory size in bytes.
- (24) `rss` %ld  
Resident Set Size: number of pages the process has in real memory. This is just the pages which count toward text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out.
- (25) `rsslim` %lu

Current soft limit in bytes on the rss of the process; see the description of `RLIMIT_RSS` in `getrlimit(2)`.

- (26) `startcode %lu [PT]`  
The address above which program text can run.
- (27) `endcode %lu [PT]`  
The address below which program text can run.
- (28) `startstack %lu [PT]`  
The address of the start (i.e., bottom) of the stack.
- (29) `kstkesp %lu [PT]`  
The current value of ESP (stack pointer), as found in the kernel stack page for the process.
- (30) `kstkeip %lu [PT]`  
The current EIP (instruction pointer).
- (31) `signal %lu`  
The bitmap of pending signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use `/proc/[pid]/status` instead.
- (32) `blocked %lu`  
The bitmap of blocked signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use `/proc/[pid]/status` instead.
- (33) `sigignore %lu`  
The bitmap of ignored signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use `/proc/[pid]/status` instead.
- (34) `sigcatch %lu`  
The bitmap of caught signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use `/proc/[pid]/status` instead.
- (35) `wchan %lu [PT]`  
This is the "channel" in which the process is waiting. It is the address of a location in the kernel where the process is sleeping. The corresponding symbolic name can be found in `/proc/[pid]/wchan`.
- (36) `nswap %lu`  
Number of pages swapped (not maintained).
- (37) `cnswap %lu`  
Cumulative nswap for child processes (not maintained).
- (38) `exit_signal %d (since Linux 2.1.22)`  
Signal to be sent to parent when we die.
- (39) `processor %d (since Linux 2.2.8)`  
CPU number last executed on.
- (40) `rt_priority %u (since Linux 2.5.19)`  
Real-time scheduling priority, a number in the range 1 to 99 for processes scheduled under a real-time policy, or 0, for non-real-time processes (see `sched_setscheduler(2)`).
- (41) `policy %u (since Linux 2.5.19)`  
Scheduling policy (see `sched_setscheduler(2)`). Decode using the `SCHED_*` constants in `linux/sched.h`.  
  
The format for this field was `%lu` before Linux 2.6.22.
- (42) `delayacct_blkio_ticks %llu (since Linux 2.6.18)`  
Aggregated block I/O delays, measured in clock ticks (centiseconds).
- (43) `guest_time %lu (since Linux 2.6.24)`  
Guest time of the process (time spent running a virtual CPU for a guest operating system), measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).

- (44) `cguest_time %ld` (since Linux 2.6.24)  
 Guest time of the process's children, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- (45) `start_data %lu` (since Linux 3.3) [PT]  
 Address above which program initialized and uninitialized (BSS) data are placed.
- (46) `end_data %lu` (since Linux 3.3) [PT]  
 Address below which program initialized and uninitialized (BSS) data are placed.
- (47) `start_brk %lu` (since Linux 3.3) [PT]  
 Address above which program heap can be expanded with `brk(2)`.
- (48) `arg_start %lu` (since Linux 3.5) [PT]  
 Address above which program command-line arguments (`argv`) are placed.
- (49) `arg_end %lu` (since Linux 3.5) [PT]  
 Address below program command-line arguments (`argv`) are placed.
- (50) `env_start %lu` (since Linux 3.5) [PT]  
 Address above which program environment is placed.
- (51) `env_end %lu` (since Linux 3.5) [PT]  
 Address below which program environment is placed.
- (52) `exit_code %d` (since Linux 3.5) [PT]  
 The thread's exit status in the form reported by `waitpid(2)`.

```
readProcStats :: ProcessID → IO [Counter]
```

```
readProcStats pid = do
```

```
  ps0 ← readProcList (pathProcStat pid)
```

```
  let ps = zip colnames ps0
```

```
      psUseful = filter (("unused" ≠) ∘ fst) ps
```

```
  return $ map (λ(n,i) → Counter StatInfo n (PureI i)) psUseful
```

```
where
```

```
  colnames :: [Text]
```

```
  colnames = [ "pid", "unused", "unused", "ppid", "pgrp", "session", "ttynr", "tpgid", "flags", "minfl",
    , "cminflt", "majflt", "cmajflt", "utime", "stime", "cutime", "cstime", "priority", "nice", "num",
    , "itrealvalue", "starttime", "vsize", "rss", "rsslim", "startcode", "endcode", "startstack",
    , "signal", "blocked", "sigignore", "sigcatch", "wchan", "nswap", "cnswap", "exitsignal", "proc",
    , "policy", "blkio", "guesttime", "cguesttime", "startdata", "enddata", "startbrk", "argstart",
    , "envend", "exitcode"
  ]
```

```
readProcIO - //proc//<pid >//io
```

```
/proc/[pid]/io (since kernel 2.6.20)
```

This file contains I/O statistics for the process, for example:

```
# cat /proc/3828/io
rchar: 323934931
wchar: 323929600
syscr: 632687
syscw: 632675
read_bytes: 0
write_bytes: 323932160
cancelled_write_bytes: 0
```

The fields are as follows:

```
rchar: characters read
    The number of bytes which this task has caused to be read from storage. This is simply the sum
    of bytes which this process passed to read(2) and similar system calls. It includes things such
    as terminal I/O and is unaffected by whether or not actual physical disk I/O was required (the
    read might have been satisfied from pagecache).

wchar: characters written
    The number of bytes which this task has caused, or shall cause to be written to disk. Similar
    caveats apply here as with rchar.

syscr: read syscalls
    Attempt to count the number of read I/O operations—that is, system calls such as read(2) and
    pread(2).

syscw: write syscalls
    Attempt to count the number of write I/O operations—that is, system calls such as write(2) and
    pwrite(2).

read_bytes: bytes read
    Attempt to count the number of bytes which this process really did cause to be fetched from the
    storage layer. This is accurate for block-backed filesystems.

write_bytes: bytes written
    Attempt to count the number of bytes which this process caused to be sent to the storage layer.

cancelled_write_bytes:
    The big inaccuracy here is truncate. If a process writes 1MB to a file and then deletes the
    file, it will in fact perform no writeout. But it will have been accounted as having caused 1MB
    of write. In other words: this field represents the number of bytes which this process caused
    to not happen, by truncating pagecache. A task can cause "negative" I/O too. If this task
    truncates some dirty pagecache, some I/O which another task has been accounted for (in its
    write\_bytes) will not be happening.
```

Note: In the current implementation, things are a bit racy on 32-bit systems: if process A reads process B's `/proc/[pid]/io` while process B is updating one of these 64-bit counters, process A could see an intermediate result.

Permission to access this file is governed by a ptrace access mode `PTRACE\_MODE\_READ\_FSCREDS` check; see `ptrace(2)`.

*readProcIO :: ProcessID → IO [Counter]*

*readProcIO pid = do*

*ps0 ← readProcList (pathProcIO pid)*

*let ps = zip3 colnames ps0 units*

*return \$ map (λ(n,i,u) → Counter IOCounter n (u i)) ps*

**where**

*colnames :: [Text]*

*colnames = [ "rchar", "wchar", "syscr", "syscw", "rbytes", "wbytes", "cxwbytes" ]*

*units = [ Bytes ∘ fromInteger, Bytes ∘ fromInteger, PureI, PureI, Bytes ∘ fromInteger, Bytes ∘ fromInteger, Bytes ∘ fromInteger ]*

## Network TCP/IP counters

```
example:
\\
cat /proc/<pid>/net/netstat
\\
TcpExt: SyncookiesSent SyncookiesRecv SyncookiesFailed EmbryonicRsts PruneCalled RcvPruned OfoPruned OutOfWindowIcmps Lo!
!ckDroppedIcmps ArpFilter TW TwRecycled TwKilled PAWSActive PAWSEstab DelayedACKs DelayedACKLocked DelayedACKLost Listen0!
!verflows ListenDrops TCPHPHits TCPPureAcks TCPHPAcks TCPReNoRecovery TCPSackRecovery TCPSACKReneging TCPSACKReorder TCPR!
```

```
!enoReorder TCPTSReorder TCPFullUndo TCPPartialUndo TCPDSACKUndo TCPLossUndo TCPLostRetransmit TCPReorderFailures TCPSackFail!
!ures TCPLossFailures TCPFastRetrans TCPSlowStartRetrans TCPTimeouts TCPLossProbes TCPLossProbeRecovery TCPReorderRecoveryF!
!ail TCPSackRecoveryFail TCPRecvCollapsed TCPDSACKOldSent TCPDSACKOfoSent TCPDSACKRecv TCPDSACKOfoRecv TCPAbortOnData TCPA!
!bortOnClose TCPAbortOnMemory TCPAbortOnTimeout TCPAbortOnLinger TCPAbortFailed TCPMemoryPressures TCPMemoryPressuresChro!
!no TCPSACKDiscard TCPSACKIgnoredOld TCPSACKIgnoredNoUndo TCPSpuriousRTos TCPMD5NotFound TCPMD5Unexpected TCPMD5Failure!
! TCPSackShifted TCPSackMerged TCPSackShiftFallback TCPBacklogDrop PFMemallocDrop TCPMinTTLDrop TCPDeferAcceptDrop IPReve!
!rsePathFilter TCPTimeWaitOverflow TCPReqQFullDoCookies TCPReqQFullDrop TCPRetransFail TCPRecvCoalesce TCPOfOQueue TCPOfOD!
!rop TCPOfOMerge TCPChallengeACK TCPSYNChallenge TCPFastOpenActive TCPFastOpenActiveFailTCPFastOpenPassive TCPFastOpenPas!
!siveFail TCPFastOpenListenOverflow TCPFastOpenCookieReqd TCPFastOpenBlackhole TCPSpuriousRtxHostQueues BusyPollRxBpackets!
! TCPAutoCorking TCPFromZeroWindowAdv TCPToZeroWindowAdv TCPWantZeroWindowAdv TCPSynRetrans TCPOrigDataSent TCPHystartTra!
!inDetect TCPHystartTrainCwnd TCPHystartDelayDetect TCPHystartDelayCwnd TCPACKSkippedSynRecv TCPACKSkippedPAWS TCPACKSkip!
!pedSeq TCPACKSkippedFinWait2 TCPACKSkippedTimeWait TCPACKSkippedChallenge TCPWinProbe TCPKeepAlive TCPMTUPFail TCPMTUPSu!
!ccess TCPDelivered TCPDeliveredCE TCPAckCompressed
TcpExt: 0 0 0 0 28 0 0 0 0 1670 1 0 0 6 6029 1 1766 0 0 384612 66799 105553 0 21 0 638 0 1 7 1 1 32 128 0 1 0 22 0 116!
! 383 19 0 0 0 1788 224 178 0 435 224 0 13 0 0 0 0 0 67 0 0 0 3 1 668 0 0 0 4 0 0 0 0 0 91870 4468 0 224 22 23 0 0 0 !
!0 0 0 0 6 0 21492 0 0 11 188 188680 6 145 13 425 0 3 4 0 0 1 117 22984 0 0 192495 0 4500
IpExt: InNoRoutes InTruncatedPkts InMcastPkts OutMcastPkts InBcastPkts OutBcastPkts InOctets OutOctets InMcastOctets Out!
!McastOctets InBcastOctets OutBcastOctets InCsumErrors InNoECTPkts InECT1Pkts InECT0Pkts InCEPkts
IpExt: 0 0 20053 8977 2437 23 3163525943 196480057 2426648 1491754 394285 5523 0 3513269 0 217426 0
```

```
readProcNet :: ProcessID → IO [Counter]
readProcNet pid = do
  ls0 ← lines < $ > readFile (pathProcNet pid)
  let ps0 = readinfo ls0
  let ps1 = map (λ(n,c) → (n, readMaybe c :: Maybe Integer)) ps0
  return $ mapCounters $ filter selcolumns ps1
where
  construct "IpExt:OutOctets" i = Bytes $ fromInteger i
  construct "IpExt:InOctets" i = Bytes $ fromInteger i
  construct _i = PureI i
  -- only a few selected columns will be returned
  selcolumns (n,_) = n ∈ [ "IpExt:OutOctets", "IpExt:InOctets" ]
  mapCounters [] = []
  mapCounters ((n,c):r) = case c of
    Nothing → mapCounters r
    Just i → mapCounters r <> [Counter NetCounter (pack n) (construct n i)]
  readinfo :: [String] → [(String,String)]
  readinfo [] = []
  readinfo (_:[]) = []
  readinfo (l1:l2:r) =
    let col0 = words l1
        cols = tail col0
        vals = tail $ words l2
        pref = head col0
    in
      readinfo r <> zip (map (λn → pref ++ n) cols) vals
```

### 1.5.10 Cardano.BM.Data.Aggregated

#### Measurable

A **Measurable** may consist of different types of values. Time measurements are strict, so are *Bytes* which are externally measured. The real or integral numeric values are lazily linked, so we can decide later to drop them.

```

data Measurable = Microseconds {-# UNPACK #-} !Word64
  | Nanoseconds {-# UNPACK #-} !Word64
  | Seconds      {-# UNPACK #-} !Word64
  | Bytes        {-# UNPACK #-} !Word64
  | PureD        Double
  | PureI        Integer
  | Severity     S.Severity
deriving (Eq, Read, Generic, ToJSON)

```

**Measurable** can be transformed to an integral value.

**instance** Ord **Measurable** **where**

```

compare (Seconds a) (Seconds b)           = compare a b
compare (Microseconds a) (Microseconds b) = compare a b
compare (Nanoseconds a) (Nanoseconds b)   = compare a b
compare (Seconds a) (Microseconds b)      = compare (a * 1000000) b
compare (Nanoseconds a) (Microseconds b)  = compare a (b * 1000)
compare (Seconds a) (Nanoseconds b)       = compare (a * 1000000000) b
compare (Microseconds a) (Nanoseconds b)  = compare (a * 1000) b
compare (Microseconds a) (Seconds b)       = compare a (b * 1000000)
compare (Nanoseconds a) (Seconds b)       = compare a (b * 1000000000)
compare (Bytes a) (Bytes b)                = compare a b
compare (PureD a) (PureD b)                = compare a b
compare (PureI a) (PureI b)                = compare a b
compare (Severity a) (Severity b)           = compare a b
compare (PureI a) (Seconds b)              | a ≥ 0 = compare a (toInteger b)
compare (PureI a) (Microseconds b)         | a ≥ 0 = compare a (toInteger b)
compare (PureI a) (Nanoseconds b)          | a ≥ 0 = compare a (toInteger b)
compare (PureI a) (Bytes b)                | a ≥ 0 = compare a (toInteger b)
compare (Seconds a) (PureI b)              | b ≥ 0 = compare (toInteger a) b
compare (Microseconds a) (PureI b)         | b ≥ 0 = compare (toInteger a) b
compare (Nanoseconds a) (PureI b)         | b ≥ 0 = compare (toInteger a) b
compare (Bytes a) (PureI b)               | b ≥ 0 = compare (toInteger a) b
compare a@(PureD _) (PureI b)              = compare (getInteger a) b
compare (PureI a) b@(PureD _)              = compare a (getInteger b)
compare a b                                = error $ "cannot compare " ++ (showType a) ++ " " ++ (show a) ++ " "

```

**Measurable** can be transformed to an integral value.

```

getInteger :: Measurable → Integer
getInteger (Microseconds a) = toInteger a
getInteger (Nanoseconds a) = toInteger a
getInteger (Seconds a)     = toInteger a
getInteger (Bytes a)       = toInteger a
getInteger (PureI a)       = a
getInteger (PureD a)       = round a
getInteger (Severity a)    = toInteger (fromEnum a)

```

**Measurable** can be transformed to a rational value.



```

getDouble :: Measurable → Double
getDouble (Microseconds a) = fromIntegral a
getDouble (Nanoseconds a) = fromIntegral a
getDouble (Seconds a)      = fromIntegral a
getDouble (Bytes a)        = fromIntegral a
getDouble (PureI a)        = fromIntegral a
getDouble (PureD a)        = a
getDouble (Severity a)     = fromIntegral (fromEnum a)

```

It is a numerical value, thus supports functions to operate on numbers.

**instance Num Measurable where**

```

(+) (Microseconds a) (Microseconds b) = Microseconds (a + b)
(+) (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a + b)
(+) (Seconds a)      (Seconds b)      = Seconds      (a + b)
(+) (Bytes a)        (Bytes b)        = Bytes        (a + b)
(+) (PureI a)        (PureI b)        = PureI        (a + b)
(+) (PureD a)        (PureD b)        = PureD        (a + b)
(+) --              = error "Trying to add values with different units"

(*) (Microseconds a) (Microseconds b) = Microseconds (a * b)
(*) (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a * b)
(*) (Seconds a)      (Seconds b)      = Seconds      (a * b)
(*) (Bytes a)        (Bytes b)        = Bytes        (a * b)
(*) (PureI a)        (PureI b)        = PureI        (a * b)
(*) (PureD a)        (PureD b)        = PureD        (a * b)
(*) --              = error "Trying to multiply values with different units"

abs (Microseconds a) = Microseconds (abs a)
abs (Nanoseconds a) = Nanoseconds (abs a)
abs (Seconds a)      = Seconds      (abs a)
abs (Bytes a)        = Bytes        (abs a)
abs (PureI a)        = PureI        (abs a)
abs (PureD a)        = PureD        (abs a)
abs (Severity _)     = error "cannot compute absolute value for Severity"

signum (Microseconds a) = Microseconds (signum a)
signum (Nanoseconds a) = Nanoseconds (signum a)
signum (Seconds a)      = Seconds      (signum a)
signum (Bytes a)        = Bytes        (signum a)
signum (PureI a)        = PureI        (signum a)
signum (PureD a)        = PureD        (signum a)
signum (Severity _)     = error "cannot compute sign of Severity"

negate (Microseconds a) = Microseconds (negate a)
negate (Nanoseconds a) = Nanoseconds (negate a)
negate (Seconds a)      = Seconds      (negate a)
negate (Bytes a)        = Bytes        (negate a)
negate (PureI a)        = PureI        (negate a)
negate (PureD a)        = PureD        (negate a)
negate (Severity _)     = error "cannot negate Severity"

fromIntegral = PureI

```

Pretty printing of **Measurable**.

```
instance Show Measurable where
  show (Microseconds a) = show a
  show (Nanoseconds a)  = show a
  show (Seconds a)       = show a
  show (Bytes a)         = show a
  show (PureI a)         = show a
  show (PureD a)         = show a
  show (Severity a)      = show a

showUnits :: Measurable → String
showUnits (Microseconds _) = " μs"
showUnits (Nanoseconds _) = " ns"
showUnits (Seconds _)     = " s"
showUnits (Bytes _)       = " B"
showUnits (PureI _)       = ""
showUnits (PureD _)       = ""
showUnits (Severity _)   = ""

showType :: Measurable → String
showType (Microseconds _) = "Microseconds"
showType (Nanoseconds _) = "Nanoseconds"
showType (Seconds _)     = "Seconds"
showType (Bytes _)       = "Bytes"
showType (PureI _)       = "PureI"
showType (PureD _)       = "PureD"
showType (Severity _)   = "Severity"

-- show in S.I. units
showSI :: Measurable → String
showSI (Microseconds a) = show (fromFloatDigits ((fromIntegral a) / (1000000 :: Float))) ++
  showUnits (Seconds a)
showSI (Nanoseconds a) = show (fromFloatDigits ((fromIntegral a) / (1000000000 :: Float))) ++
  showUnits (Seconds a)
showSI v@(Seconds a)   = show a ++ showUnits v
showSI v@(Bytes a)     = show a ++ showUnits v
showSI v@(PureI a)     = show a ++ showUnits v
showSI v@(PureD a)     = show a ++ showUnits v
showSI v@(Severity a) = show a ++ showUnits v
```

## Stats

A **Stats** statistics is strictly computed.

```
data BaseStats = BaseStats {
  fmin :: !Measurable,
  fmax :: !Measurable,
  fcount :: {-# UNPACK #-} !Int,
  fsum_A :: {-# UNPACK #-} !Double,
```

```

fsum_B :: {-# UNPACK #-} !Double
} deriving (Generic, ToJSON, Show)

instance Eq BaseStats where
  (BaseStats mina maxa counta sumAa sumBa) ≡ (BaseStats minb maxb countb sumAb sumBb) =
    mina ≡ minb ∧ maxa ≡ maxb ∧ counta ≡ countb ∧
    abs (sumAa - sumAb) < 1.0e-4 ∧
    abs (sumBa - sumBb) < 1.0e-4

data Stats = Stats {
  flast :: !Measurable,
  fold :: !Measurable,
  fbasic :: !BaseStats,
  fdelta :: !BaseStats,
  ftimed :: !BaseStats
} deriving (Eq, Generic, ToJSON, Show)

meanOfStats :: BaseStats → Double
meanOfStats = fsum_A

stdevOfStats :: BaseStats → Double
stdevOfStats s =
  if fcount s < 2
  then 0
  else sqrt $ (fsum_B s) / (fromInteger $ fromIntegral (fcount s) - 1)

```

**instance Semigroup Stats** disabled for the moment, because not needed.

We use a parallel algorithm to update the estimation of mean and variance from two sample statistics. (see [https://en.wikipedia.org/wiki/Algorithms\\_for\\_calculating\\_variance#Parallel\\_algorithm](https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorithm))

```

instance Semigroup Stats where
  (<>) a b = let counta = fcount a
              countb = fcount b
              newcount = counta + countb
              delta = fsum_A b - fsum_A a
            in
  Stats {flast = flast b -- right associative
        ,fmin   = min (fmin a) (fmin b)
        ,fmax   = max (fmax a) (fmax b)
        ,fcount = newcount
        ,fsum_A = fsum_A a + (delta / fromInteger newcount)
        ,fsum_B = fsum_B a + fsum_B b + (delta * delta) * (fromInteger (counta * countb) / fromInteger newcount)
        }

stats2Text :: Stats → Text
stats2Text (Stats slast _ sbasic sdelta stimed) =
  pack $
    "{ last=" ++ show slast ++

```

```

", basic-stats=" ++ showStats' (sbasic) ++
", delta-stats=" ++ showStats' (sdelta) ++
", timed-stats=" ++ showStats' (stimed) ++
" }"

```

where

```

showStats' :: BaseStats → String
showStats' s =
  ", { min=" ++ show (fmin s) ++
  ", max=" ++ show (fmax s) ++
  ", mean=" ++ show (meanOfStats s) ++ showUnits (fmin s) ++
  ", std-dev=" ++ show (stdevOfStats s) ++
  ", count=" ++ show (fcount s) ++
  " }"

```

### Exponentially Weighted Moving Average (EWMA)

Following [https://en.wikipedia.org/wiki/Moving\\_average#Exponential\\_moving\\_average](https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average) we calculate the exponential moving average for a series of values  $Y_t$  according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

```

data EWMA = EmptyEWMA {alpha :: Double}
  | EWMA {alpha :: Double
    , avg :: Measurable
    } deriving (Show, Eq, Generic, ToJSON)

```

### Aggregated

```

data Aggregated = AggregatedStats Stats
  | AggregatedEWMA EWMA
  deriving (Eq, Generic, ToJSON)

```

**instance Semigroup Aggregated** disabled for the moment, because not needed.

```

instance Semigroup Aggregated where
  (<>) (AggregatedStats a) (AggregatedStats b) =
    AggregatedStats (a <> b)
  (<>) _ _ = error "Cannot combine different objects"

```

```

singletonStats :: Measurable → Aggregated

```

```

singletonStats a =

```

```

  let stats = Stats {flast = a
    , fold      = Nanoseconds 0
    , fbasic    = BaseStats

```

```

    {fmin = a
    ,fmax = a
    ,fcount = 1
    ,fsum_A = getDouble a
    ,fsum_B = 0}
  ,fdelta = BaseStats
    {fmin = 0
    ,fmax = 0
    ,fcount = 0
    ,fsum_A = 0
    ,fsum_B = 0}
  ,ftimed = BaseStats
    {fmin = Nanoseconds 999999999999
    ,fmax = Nanoseconds 0
    ,fcount = (-1)
    ,fsum_A = 0
    ,fsum_B = 0}
  }
in
  AggregatedStats stats

```

```

instance Show Aggregated where
  show (AggregatedStats astats) =
    "{ stats = " ++ show astats ++ " }"
  show (AggregatedEWMA a) = show a

```

### 1.5.11 Cardano.BM.Data.AggregatedKind

#### AggregatedKind

This identifies the type of Aggregated.

```

data AggregatedKind = StatsAK
  | EwmaAK {alpha :: Double}
  deriving (Generic, Eq, Show, FromJSON, ToJSON, Read)

```

### 1.5.12 Cardano.BM.Data.Backend

#### Accepts a NamedLogItem

Instances of this type class accept a **NamedLogItem** and deal with it.

```

class IsEffectuator t where
  effectuate :: t → NamedLogItem → IO ()
  effectuatefrom :: forall s. (IsEffectuator s) ⇒ t → NamedLogItem → s → IO ()
  default effectuatefrom :: forall s. (IsEffectuator s) ⇒ t → NamedLogItem → s → IO ()
  effectuatefrom t nli _ = effectuate t nli
  handleOverflow :: t → IO ()

```

### Declaration of a Backend

A backend is life-cycle managed, thus can be *realized* and *unrealized*.

```
class (IsEffectuator t) ⇒ IsBackend t where
  typeof    :: t → BackendKind
  realize    :: Configuration → IO t
  realizefrom :: forall s o (IsEffectuator s) ⇒ Trace IO → s → IO t
  default realizefrom :: forall s o (IsEffectuator s) ⇒ Trace IO → s → IO t
  realizefrom (ctx, _) _ = realize (configuration ctx)
  unrealize :: t → IO ()
```

### Backend

This data structure for a backend defines its behaviour as an **IsEffectuator** when processing an incoming message, and as an **IsBackend** for unrealizing the backend.

```
data Backend = MkBackend
  { bEffectuate :: NamedLogItem → IO ()
  , bUnrealize :: IO ()
  }
```

#### 1.5.13 Cardano.BM.Data.BackendKind

##### BackendKind

This identifies the backends that can be attached to the **Switchboard**.

```
data BackendKind =
  AggregationBK
  | EKGViewBK
  | KatipBK
  | MonitoringBK
  | SwitchboardBK
  deriving (Generic, Eq, Ord, Show, ToJSON, FromJSON, Read)
```

#### 1.5.14 Cardano.BM.Data.Configuration

Data structure to help parsing configuration files.

##### Representation

```
type Port = Int
data Representation = Representation
  { minSeverity    :: Severity
  , rotation       :: Maybe RotationParameters
  , setupScribes   :: [ScribeDefinition]
  , defaultScribes :: [(ScribeKind, Text)]
```

```

,setupBackends :: [BackendKind]
,defaultBackends :: [BackendKind]
,hasEKG        :: Maybe Port
,hasGUI        :: Maybe Port
,options        :: HM.HashMap Text Object
}
deriving (Generic, Show, ToJSON, FromJSON)

```

## parseRepresentation

```

parseRepresentation :: FilePath → IO Representation
parseRepresentation fp = do
  repr :: Representation ← decodeFileThrow fp
  return $ implicit_fill_representation repr

```

after parsing the configuration representation we implicitly correct it.

```

implicit_fill_representation :: Representation → Representation
implicit_fill_representation =
  remove_ekgview_if_not_defined ∘
  filter_duplicates_from_backends ∘
  filter_duplicates_from_scribes ∘
  union_setup_and_usage_backends ∘
  add_ekgview_if_port_defined ∘
  add_katip_if_any_scribes
where
  filter_duplicates_from_backends r =
    r {setupBackends = mkUniq $ setupBackends r}
  filter_duplicates_from_scribes r =
    r {setupScribes = mkUniq $ setupScribes r}
  union_setup_and_usage_backends r =
    r {setupBackends = setupBackends r <> defaultBackends r}
# ifdef ENABLE_EKG
  remove_ekgview_if_not_defined r =
    case hasEKG r of
      Nothing → r {defaultBackends = filter (λbk → bk ≠ EKGViewBK) (defaultBackends r)
                  , setupBackends = filter (λbk → bk ≠ EKGViewBK) (setupBackends r)
                  }
      Just _ → r
  add_ekgview_if_port_defined r =
    case hasEKG r of
      Nothing → r
      Just _ → r {setupBackends = setupBackends r <> [EKGViewBK]}
# else
  remove_ekgview_if_not_defined = id
  add_ekgview_if_port_defined = id
# endif

```

```

add_katip_if_any_scribes r =
  if (any  $\neg$  [null $ setupScribes r, null $ defaultScribes r])
  then r {setupBackends = setupBackends r <> [KatipBK]}
  else r
mkUniq :: Ord a  $\Rightarrow$  [a]  $\rightarrow$  [a]
mkUniq = Set.toList  $\circ$  Set.fromList

```

### 1.5.15 Cardano.BM.Data.Counter

#### Counter

```

data Counter = Counter
  { cType :: CounterType
  , cName :: Text
  , cValue :: Measurable
  }
  deriving (Eq, Show, Generic, ToJSON)

data CounterType = MonotonicClockTime
  | MemoryCounter
  | StatInfo
  | IOCounter
  | NetCounter
  | CpuCounter
  | RTStats
  deriving (Eq, Show, Generic, ToJSON)

instance ToJSON Microsecond where
  toJSON = toJSON  $\circ$  toMicroseconds
  toEncoding = toEncoding  $\circ$  toMicroseconds

```

#### Names of counters

```

nameCounter :: Counter  $\rightarrow$  Text
nameCounter (Counter MonotonicClockTime _) = "Time-interval"
nameCounter (Counter MemoryCounter _) = "Mem"
nameCounter (Counter StatInfo _) = "Stat"
nameCounter (Counter IOCounter _) = "IO"
nameCounter (Counter NetCounter _) = "Net"
nameCounter (Counter CpuCounter _) = "Cpu"
nameCounter (Counter RTStats _) = "RTS"

```

#### CounterState

```

data CounterState = CounterState {
  csIdentifier :: Unique

```



```

    ,csCounters :: [Counter]
  }
  deriving (Generic, ToJSON)
instance ToJSON Unique where
  toJSON = toJSON ∘ hashUnique
  toEncoding = toEncoding ∘ hashUnique
instance Show CounterState where
  show cs = (show ∘ hashUnique) (csIdentifier cs)
    <> " => " <> (show $ csCounters cs)

```

### Difference between counters

```

diffCounters :: [Counter] → [Counter] → [Counter]
diffCounters openings closings =
  getCountersDiff openings closings
where
  getCountersDiff :: [Counter]
    → [Counter]
    → [Counter]
  getCountersDiff as bs =
    let
      getName counter = nameCounter counter <> cName counter
      asNames = map getName as
      aPairs = zip asNames as
      bsNames = map getName bs
      bs' = zip bsNames bs
      bPairs = HM.fromList bs'
    in
      catMaybes $ (flip map) aPairs $ λ(name, Counter _ _ startValue) →
        case HM.lookup name bPairs of
          Nothing → Nothing
          Just counter → let endValue = cValue counter
                        in Just counter {cValue = endValue − startValue}

```

#### 1.5.16 Cardano.BM.Data.LogItem

##### LoggerName

A **LoggerName** has currently type *Text*.

```
type LoggerName = Text
```

##### NamedLogItem

```
type NamedLogItem = LogNamed LogObject
```

## LogNamed

A **LogNamed** contains of a context name and some log item.

```
data LogNamed item = LogNamed
  { lnName :: LoggerName
  , lnItem :: item
  } deriving (Show)
deriving instance Generic item => Generic (LogNamed item)
deriving instance (ToJSON item, Generic item) => ToJSON (LogNamed item)
```

## Logging of outcomes with LogObject

```
data LogObject = LogObject LOMeta LOContent
  deriving (Generic, Show, ToJSON)
```

Meta data for a **LogObject**. Text was selected over ThreadId in order to be able to use the logging system under SimM of ouroboros-network because ThreadId from Control.Concurrent lacks a Read instance.

```
data LOMeta = LOMeta {
  tstamp :: {-# UNPACK #-} !UTCTime
  , tid    :: {-# UNPACK #-} !Text
  , severity :: !Severity
}
deriving (Show)
instance ToJSON LOMeta where
  toJSON (LOMeta tstamp tid sev) =
    object [ "tstamp" . = tstamp, "tid" . = show tid, "severity" . = show sev ]
mkLOMeta :: Severity -> IO LOMeta
mkLOMeta sev =
  LOMeta <$> get currentTime
    <*> (pack o show <$> myThreadId)
    <*> pure sev
```

Payload of a **LogObject**:

```
data LOContent = LogMessage LogItem
  | LogValue Text Measurable
  | ObserveOpen CounterState
  | ObserveDiff CounterState
  | ObserveClose CounterState
  | AggregatedMessage [(Text, Aggregated)]
  | MonitoringEffect LogObject
  | KillPill
  deriving (Generic, Show, ToJSON)
```

**LogItem**

TODO `liPayload :: ToObject`

```
data LogItem = LogItem
  { liSelection :: LogSelection
  , liPayload :: Text -- TODO should become ToObject
  } deriving (Show, Generic, ToJSON)
```

```
data LogSelection =
  Private -- only to private logs.
  | Both -- to public and private logs.
  deriving (Show, Generic, ToJSON, FromJSON)
```

**1.5.17 Cardano.BM.Data.Observable****ObservableInstance**

```
data ObservableInstance = MonotonicClock
  | MemoryStats
  | ProcessStats
  | NetStats
  | IOStats
  | GhcRtsStats
  deriving (Generic, Eq, Show, FromJSON, ToJSON, Read)
```

**1.5.18 Cardano.BM.Data.Output****OutputKind**

```
data OutputKind = TVarList (STM.TVar [LogObject])
  | TVarListNamed (STM.TVar [LogNamed LogObject])
  deriving (Eq)
```

**ScribeKind**

This identifies katip's scribes by type.

```
data ScribeKind = FileTextSK
  | FileJsonSK
  | StdoutSK
  | StderrSK
  deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON)
```

### ScribeId

A scribe is identified by **ScribeKind** x *Filename*

```
type ScribeId = Text -- (ScribeKind :: Filename)
```

### ScribePrivacy

This declares if a scribe will be public (and must not contain sensitive data) or private.

```
data ScribePrivacy = ScPublic | ScPrivate
deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON)
```

### ScribeDefinition

This identifies katip's scribes by type.

```
data ScribeDefinition = ScribeDefinition
  { scKind    :: ScribeKind
  , scName    :: Text
  , scPrivacy :: ScribePrivacy
  , scRotation :: Maybe RotationParameters
  }
deriving (Generic, Eq, Ord, Show, ToJSON)
instance FromJSON ScribeDefinition where
  parseJSON (Object o) = do
    kind    <- o .: "scKind"
    name    <- o .: "scName"
    mayPrivacy <- o .:? "scPrivacy"
    rotation <- o .:? "scRotation"
    return $ ScribeDefinition
      { scKind    = kind
      , scName    = name
      , scPrivacy = fromMaybe ScPublic mayPrivacy
      , scRotation = rotation
      }
    parseJSON invalid = typeMismatch "ScribeDefinition" invalid
```

## 1.5.19 Cardano.BM.Data.Rotation

### RotationParameters

```
data RotationParameters = RotationParameters
  { rpLogLimitBytes :: !Word64 -- max size of file in bytes
  , rpMaxAgeHours   :: !Word   -- hours
  , rpKeepFilesNum :: !Word   -- number of files to keep
  } deriving (Generic, Show, Eq, Ord, FromJSON, ToJSON)
```

### 1.5.20 Cardano.BM.Data.Severity

#### Severity

The intended meaning of severity codes:

Debug *detailed information about values and decision flow* **Info** general information of events; progressing properly **Notice** *needs attention; something  $\rightarrow$  progressing properly* **Warning** may continue into an error condition if continued **Error** *unexpected set of event or condition occurred* **Critical** error condition causing degrade of operation **Alert** *a subsystem is no longer operating correctly, likely requires manual* at this point, the system can never progress without additional intervention

We were informed by the Syslog taxonomy: [https://en.wikipedia.org/wiki/Syslog#Severity\\_level](https://en.wikipedia.org/wiki/Syslog#Severity_level)

```
data Severity = Debug
  | Info
  | Notice
  | Warning
  | Error
  | Critical
  | Alert
  | Emergency
  deriving (Show, Eq, Ord, Enum, Generic, ToJSON, Read)

instance FromJSON Severity where
  parseJSON = withText "severity" $ \case
    "Debug"    → pure Debug
    "Info"     → pure Info
    "Notice"   → pure Notice
    "Warning"  → pure Warning
    "Error"    → pure Error
    "Critical" → pure Critical
    "Alert"    → pure Alert
    "Emergency" → pure Emergency
    _          → pure Info-- catch all
```

### 1.5.21 Cardano.BM.Data.SubTrace

#### SubTrace

```
data NameSelector = Exact Text | StartsWith Text | EndsWith Text | Contains Text
  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)

data DropName = Drop NameSelector
  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)

data UnhideNames = Unhide [NameSelector]
  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)

data SubTrace = Neutral
  | UntimedTrace
  | NoTrace
  | TeeTrace LoggerName
  | FilterTrace [(DropName, UnhideNames)]
```

```

| DropOpening
| ObservableTrace [ ObservableInstance ]
  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)

```

### 1.5.22 Cardano.BM.Data.Trace

#### Trace

A **Trace** consists of a **TraceContext** and a **TraceNamed** in *m*.

```
type Trace m = (TraceContext, TraceNamed m)
```

#### TraceNamed

A **TraceNamed** is a specialized **Contravariant** of type **NamedLogItem**, a **LogNamed** with payload **LogObject**.

```
type TraceNamed m = BaseTrace m (NamedLogItem)
```

#### TraceContext

We keep the context's name and a reference to the **Configuration** in the **TraceContext**.

```

data TraceContext = TraceContext
  { loggerName :: LoggerName
  , configuration :: Configuration
  , tracetype   :: SubTrace
  , minSeverity :: Severity
  }

```

### 1.5.23 Cardano.BM.Configuration

see **Cardano.BM.Configuration.Model** for the implementation.

```

getOptionOrDefault :: CM.Configuration → Text → Text → IO (Text)
getOptionOrDefault cg name def = do
  opt ← CM.getOption cg name
  case opt of
    Nothing → return def
    Just o  → return o

```

### 1.5.24 Cardano.BM.Configuration.Model

#### Configuration.Model

```

type ConfigurationMVar = MVar ConfigurationInternal
newtype Configuration = Configuration
    {getCG :: ConfigurationMVar}
-- Our internal state; see -"Configuration model"-
data ConfigurationInternal = ConfigurationInternal
    {cgMinSeverity      :: Severity
    -- minimum severity level of every object that will be output
    ,cgMapSeverity      :: HM.HashMap LoggerName Severity
    -- severity filter per loggename
    ,cgMapSubtrace      :: HM.HashMap LoggerName SubTrace
    -- type of trace per loggename
    ,cgOptions          :: HM.HashMap Text Object
    -- options needed for tracing, logging and monitoring
    ,cgMapBackend        :: HM.HashMap LoggerName [BackendKind]
    -- backends that will be used for the specific loggename
    ,cgDefBackendKs     :: [BackendKind]
    -- backends that will be used if a set of backends for the
    -- specific loggename is not set
    ,cgSetupBackends    :: [BackendKind]
    -- backends to setup; every backend to be used must have
    -- been declared here
    ,cgMapScribe         :: HM.HashMap LoggerName [ScribeId]
    -- katip scribes that will be used for the specific loggename
    ,cgMapScribeCache    :: HM.HashMap LoggerName [ScribeId]
    -- map to cache info of the cgMapScribe
    ,cgDefScribes        :: [ScribeId]
    -- katip scribes that will be used if a set of scribes for the
    -- specific loggename is not set
    ,cgSetupScribes      :: [ScribeDefinition]
    -- katip scribes to setup; every scribe to be used must have
    -- been declared here
    ,cgMapAggregatedKind :: HM.HashMap LoggerName AggregatedKind
    -- kind of Aggregated that will be used for the specific loggename
    ,cgDefAggregatedKind :: AggregatedKind
    -- kind of Aggregated that will be used if a set of scribes for the
    -- specific loggename is not set
    ,cgMonitors          :: HM.HashMap LoggerName (MEvExpr, [MEvAction])
    ,cgPortEKG           :: Int
    -- port for EKG server
    ,cgPortGUI           :: Int
    -- port for changes at runtime (NOT IMPLEMENTED YET)
    } deriving (Show, Eq)

```

### Backends configured in the **Switchboard**

For a given context name return the list of backends configured, or, in case no such configuration exists, return the default backends.

```

getBackends :: Configuration → LoggerName → IO [BackendKind]
getBackends configuration name = do
  cg ← readMVar $ getCG configuration
  let outs = HM.lookup name (cgMapBackend cg)
  case outs of
    Nothing → return (cgDefBackendKs cg)
    Just os → return os

getDefaultBackends :: Configuration → IO [BackendKind]
getDefaultBackends configuration =
  cgDefBackendKs < $ > (readMVar $ getCG configuration)

setDefaultBackends :: Configuration → [BackendKind] → IO ()
setDefaultBackends configuration bes =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgDefBackendKs = bes}

setBackends :: Configuration → LoggerName → Maybe [BackendKind] → IO ()
setBackends configuration name be =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapBackend = HM.alter (\_ → be) name (cgMapBackend cg)}

```

### Backends to be setup by the **Switchboard**

Defines the list of **Backends** that need to be setup by the **Switchboard**.

```

setSetupBackends :: Configuration → [BackendKind] → IO ()
setSetupBackends configuration bes =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgSetupBackends = bes}

getSetupBackends :: Configuration → IO [BackendKind]
getSetupBackends configuration =
  cgSetupBackends < $ > (readMVar $ getCG configuration)

```

### Scribes configured in the **Log** backend

For a given context name return the list of scribes to output to, or, in case no such configuration exists, return the default scribes to use.

```

getScribes :: Configuration → LoggerName → IO [ScribeId]
getScribes configuration name = do
  cg ← readMVar (getCG configuration)
  (updateCache, scribes) ← do
    let defs = cgDefScribes cg
    let mapscribes = cgMapScribe cg

```



```

let find_s lname = case HM.lookup lname mapscribes of
  Nothing →
    case dropToDot lname of
      Nothing → defs
      Just lname' → find_s lname'
  Just os → os
let outs = HM.lookup name (cgMapScribeCache cg)
-- look if scribes are already cached
return $ case outs of
  -- if no cached scribes found; search the appropriate scribes that
  -- they must inherit and update the cached map
  Nothing → (True, find_s name)
  Just os → (False, os)
when updateCache $ setCachedScribes configuration name $ Just scribes
return scribes

dropToDot :: Text → Maybe Text
dropToDot ts = dropToDot' (breakOnEnd "." ts)
where
  dropToDot' (_, "") = Nothing
  dropToDot' (name', _) = Just $ dropWhileEnd (≡ ' . ') name'

getCachedScribes :: Configuration → LoggerName → IO (Maybe [ScribeId])
getCachedScribes configuration name = do
  cg ← readMVar $ getCG configuration
  return $ HM.lookup name $ cgMapScribeCache cg

setScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
setScribes configuration name scribes =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapScribe = HM.alter (\_ → scribes) name (cgMapScribe cg)}

setCachedScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
setCachedScribes configuration name scribes =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapScribeCache = HM.alter (\_ → scribes) name (cgMapScribeCache cg)}

setDefaultScribes :: Configuration → [ScribeId] → IO ()
setDefaultScribes configuration scs =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgDefScribes = scs}

```

### Scribes to be setup in the **Log** backend

Defines the list of *Scribes* that need to be setup in the **Log** backend.

```

setSetupScribes :: Configuration → [ScribeDefinition] → IO ()
setSetupScribes configuration sds =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgSetupScribes = sds}
getSetupScribes :: Configuration → IO [ScribeDefinition]

```

```
getSetupScribes configuration =
  cgSetupScribes < $ > readMVar (getCG configuration)
```

### **AggregatedKind** to define the type of measurement

For a given context name return its **AggregatedKind** or in case no such configuration exists, return the default **AggregatedKind** to use.

```
getAggregatedKind :: Configuration → LoggerName → IO AggregatedKind
getAggregatedKind configuration name = do
  cg ← readMVar $ getCG configuration
  let outs = HM.lookup name (cgMapAggregatedKind cg)
  case outs of
    Nothing → return $ cgDefAggregatedKind cg
    Just os → return $ os

setDefaultAggregatedKind :: Configuration → AggregatedKind → IO ()
setDefaultAggregatedKind configuration defAK =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgDefAggregatedKind = defAK}

setAggregatedKind :: Configuration → LoggerName → Maybe AggregatedKind → IO ()
setAggregatedKind configuration name ak =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapAggregatedKind = HM.alter (\_ → ak) name (cgMapAggregatedKind cg)}
```

### Access port numbers of EKG, GUI

```
getEKGport :: Configuration → IO Int
getEKGport configuration =
  cgPortEKG < $ > (readMVar $ getCG configuration)

setEKGport :: Configuration → Int → IO ()
setEKGport configuration port =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgPortEKG = port}

getGUIport :: Configuration → IO Int
getGUIport configuration =
  cgPortGUI < $ > (readMVar $ getCG configuration)

setGUIport :: Configuration → Int → IO ()
setGUIport configuration port =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgPortGUI = port}
```

### Options

```
getOption :: Configuration → Text → IO (Maybe Text)
getOption configuration name = do
```

```

cg ← readMVar $ getCG configuration
case HM.lookup name (cgOptions cg) of
  Nothing → return Nothing
  Just o → return $ Just $ pack $ show o

```

### Global setting of minimum severity

```

minSeverity :: Configuration → IO Severity
minSeverity configuration =
  cgMinSeverity < $ > (readMVar $ getCG configuration)
setMinSeverity :: Configuration → Severity → IO ()
setMinSeverity configuration sev =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMinSeverity = sev}

```

### Relation of context name to minimum severity

```

inspectSeverity :: Configuration → Text → IO (Maybe Severity)
inspectSeverity configuration name = do
  cg ← readMVar $ getCG configuration
  return $ HM.lookup name (cgMapSeverity cg)
setSeverity :: Configuration → Text → Maybe Severity → IO ()
setSeverity configuration name sev =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapSeverity = HM.alter (\_ → sev) name (cgMapSeverity cg)}

```

### Relation of context name to SubTrace

A new context may contain a different type of **Trace**. The function **appendName** (Enter new named context) will look up the **SubTrace** for the context's name.

```

findSubTrace :: Configuration → Text → IO (Maybe SubTrace)
findSubTrace configuration name = do
  cg ← readMVar $ getCG configuration
  return $ HM.lookup name (cgMapSubtrace cg)
setSubTrace :: Configuration → Text → Maybe SubTrace → IO ()
setSubTrace configuration name trafo =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapSubtrace = HM.alter (\_ → trafo) name (cgMapSubtrace cg)}

```

## Monitors

```

Just (
  fromList [
    ("chain.creation.block", Array [
      Object (fromList [ ("monitor", String "((time > (23 s)) Or (time < (17 s)))"]),
      Object (fromList [ ("actions", Array [
        String "AlterMinSeverity \"chain.creation\" Debug" ])]))
    ], ("#aggregation.critproc.observable", Array [
      Object (fromList [ ("monitor", String "(mean >= (42))"]),
      Object (fromList [ ("actions", Array [
        String "CreateMessage \"exceeded\" \"the observable has been too long too high!\",
        String "AlterGlobalMinSeverity Info" ])]))
    ]))

getMonitors :: Configuration → IO (HM.HashMap LoggerName (MEvExpr, [MEvAction]))
getMonitors configuration = do
  cg ← readMVar $ getCG configuration
  return (cgMonitors cg)

```

## Parse configuration from file

Parse the configuration into an internal representation first. Then, fill in **Configuration** after refinement.

```

setup :: FilePath → IO Configuration
setup fp = do
  r ← R.parseRepresentation fp
  setupFromRepresentation r

parseMonitors :: Maybe (HM.HashMap Text Value) → HM.HashMap LoggerName (MEvExpr, [MEvAction])
parseMonitors Nothing = HM.empty
parseMonitors (Just hmv) = HM.mapMaybe mkMonitor hmv

where
  mkMonitor (Array a) =
    if Vector.length a ≡ 2
    then do
      e ← mkExpression $ aVector.! 0
      as ← mkActions $ aVector.! 1
      return (e, as)
    else Nothing
  mkMonitor _ = Nothing
  mkExpression :: Value → Maybe MEvExpr
  mkExpression (Object o1) =
    case HM.lookup "monitor" o1 of
      Nothing → Nothing
      Just (String expr) → MEv.parseMaybe expr
      Just _ → Nothing
  mkExpression _ = Nothing

```

```

mkActions :: Value → Maybe [MEvAction]
mkActions (Object o2) =
  case HM.lookup "actions" o2 of
    Nothing → Nothing
    Just (Array as) → Just $ map (λ(String s) → s) $ Vector.toList as
    Just _ → Nothing
mkActions _ = Nothing

setupFromRepresentation :: R.Representation → IO Configuration
setupFromRepresentation r = do
  let mapseverities0 = HM.lookup "mapSeverity" (R.options r)
      mapbackends = HM.lookup "mapBackends" (R.options r)
      mapsubtrace = HM.lookup "mapSubtrace" (R.options r)
      mapscribes0 = HM.lookup "mapScribes" (R.options r)
      mapaggregatedkinds = HM.lookup "mapAggregatedkinds" (R.options r)
      mapmonitors = HM.lookup "mapMonitors" (R.options r)
      mapseverities = parseSeverityMap mapseverities0
      mapscribes = parseScribeMap mapscribes0
  cgreg ← newMVar $ ConfigurationInternal
    { cgMinSeverity = R.minSeverity r
    , cgMapSeverity = mapseverities
    , cgMapSubtrace = parseSubtraceMap mapsubtrace
    , cgOptions = R.options r
    , cgMapBackend = parseBackendMap mapbackends
    , cgDefBackendKs = R.defaultBackends r
    , cgSetupBackends = R.setupBackends r
    , cgMapScribe = mapscribes
    , cgMapScribeCache = mapscribes
    , cgDefScribes = r_defaultScribes r
    , cgSetupScribes = fillRotationParams (R.rotation r) (R.setupScribes r)
    , cgMapAggregatedKind = parseAggregatedKindMap mapaggregatedkinds
    , cgDefAggregatedKind = StatsAK
    , cgMonitors = parseMonitors mapmonitors
    , cgPortEKG = r_hasEKG r
    , cgPortGUI = r_hasGUI r
    }
  return $ Configuration cgreg

where
  parseSeverityMap :: Maybe (HM.HashMap Text Value) → HM.HashMap Text Severity
  parseSeverityMap Nothing = HM.empty
  parseSeverityMap (Just hmv) = HM.mapMaybe mkSeverity hmv
  where
    mkSeverity (String s) = Just (read (unpack s) :: Severity)
    mkSeverity _ = Nothing

  fillRotationParams :: Maybe RotationParameters → [ScribeDefinition] → [ScribeDefinition]
  fillRotationParams defaultRotation = map $ λsd →
    if (scKind sd ≠ StdoutSK) ∧ (scKind sd ≠ StderrSK)
    then

```

```

    sd { scRotation = maybe defaultRotation Just (scRotation sd) }
  else
    -- stdout and stderr cannot be rotated
    sd { scRotation = Nothing }
parseBackendMap Nothing = HM.empty
parseBackendMap (Just hmv) = HM.map mkBackends hmv
where
  mkBackends (Array bes) = catMaybes $ map mkBackend $ Vector.toList bes
  mkBackends _ = []
  mkBackend (String s) = Just (read (unpack s) :: BackendKind)
  mkBackend _ = Nothing
parseScribeMap Nothing = HM.empty
parseScribeMap (Just hmv) = HM.map mkScribes hmv
where
  mkScribes (Array scs) = catMaybes $ map mkScribe $ Vector.toList scs
  mkScribes (String s) = [(s :: ScribeId)]
  mkScribes _ = []
  mkScribe (String s) = Just (s :: ScribeId)
  mkScribe _ = Nothing
parseSubtraceMap :: Maybe (HM.HashMap Text Value) → HM.HashMap Text SubTrace
parseSubtraceMap Nothing = HM.empty
parseSubtraceMap (Just hmv) = HM.mapMaybe mkSubtrace hmv
where
  mkSubtrace (String s) = Just (read (unpack s) :: SubTrace)
  mkSubtrace (Object hm) = mkSubtrace' (HM.lookup "tag" hm) (HM.lookup "contents" hm)
  mkSubtrace _ = Nothing
  mkSubtrace' Nothing _ = Nothing
  mkSubtrace' _ Nothing = Nothing
  mkSubtrace' (Just (String tag)) (Just (Array cs)) =
    if tag == "ObservableTrace"
    then Just $ ObservableTrace $ map (λ(String s) → (read (unpack s) :: ObservableInstance)) $ Vector.toList cs
    else Nothing
  mkSubtrace' _ _ = Nothing
r_hasEKG repr = case (R.hasEKG repr) of
  Nothing → 0
  Just p → p
r_hasGUI repr = case (R.hasGUI repr) of
  Nothing → 0
  Just p → p
r_defaultScribes repr = map (λ(k,n) → pack (show k) <> " : " <> n) (R.defaultScribes repr)
parseAggregatedKindMap Nothing = HM.empty
parseAggregatedKindMap (Just hmv) =
  let
    listv = HM.toList hmv
    mapAggregatedKind = HM.fromList $ catMaybes $ map mkAggregatedKind listv
  in

```

```

    mapAggregatedKind
  where
    mkAggregatedKind (name, String s) = Just (name, read (unpack s) :: AggregatedKind)
    mkAggregatedKind _ = Nothing

```

### Setup empty configuration

```

empty :: IO Configuration
empty = do
  cgreg ← newMVar $ ConfigurationInternal
  {cgMinSeverity    = Debug
  ,cgMapSeverity    = HM.empty
  ,cgMapSubtrace    = HM.empty
  ,cgOptions        = HM.empty
  ,cgMapBackend     = HM.empty
  ,cgDefBackendKs   = []
  ,cgSetupBackends  = []
  ,cgMapScribe       = HM.empty
  ,cgMapScribeCache = HM.empty
  ,cgDefScribes      = []
  ,cgSetupScribes    = []
  ,cgMapAggregatedKind = HM.empty
  ,cgDefAggregatedKind = StatsAK
  ,cgMonitors        = HM.empty
  ,cgPortEKG         = 0
  ,cgPortGUI         = 0
  }
  return $ Configuration cgreg

```

### 1.5.25 Cardano.BM.Configuration.Static

#### Default configuration outputting on stdout

```

defaultConfigStdout :: IO CM.Configuration
defaultConfigStdout = do
  c ← CM.empty
  CM.setMinSeverity c Debug
  CM.setSetupBackends c [KatipBK]
  CM.setDefaultBackends c [KatipBK]
  CM.setSetupScribes c [ScribeDefinition {
    scName = "stdout"
  ,scKind = StdoutSK
  ,scPrivacy = ScPublic
  ,scRotation = Nothing
  }
  ]

```





```

        ,jsStatic                = Just "static"
        ,jsCustomHTML = Just "configuration-editor.html"
    } $ prepare config
  Async.link thd
  pure ()
else pure ()

```

```

data Cmd = Backends | Scribes | Severities | SubTrace | Aggregation
  deriving (Show, Read)

```

```

prepare :: Configuration → Window → UI ()

```

```

prepare config window = void $ do

```

```

  void $ return window # set title "IOHK logging and monitoring"

```

```

  -- editing or adding map entry

```

```

  inputKey ← UI.input #. "w3-input w3-border w3-round-large"

```

```

  inputValue ← UI.input #. "w3-input w3-border w3-round-large"

```

```

  inputMap ← UI.p #. "inputmap"

```

```

  void $ element inputKey # set UI.size "30"

```

```

  void $ element inputValue # set UI.size "60"

```

```

  outputMsg ← UI.input #. "w3-input w3-border w3-round-large"

```

```

  void $ element outputMsg # set UI.size "60"

```

```

  # set UI.enabled False

```

```

let mkPairItem :: Show t ⇒ Cmd → LoggerName → t → UI Element

```

```

  mkPairItem cmd n v =

```

```

    let entries = [ UI.td #+ [string (unpack n)]

```

```

      , UI.td #+ [string (show v)]

```

```

      , UI.td #+ [do

```

```

        b ← UI.button #. "w3-small w3-btn w3-ripple w3-teal" #+ [UI.bold #+ [string "edit"]]

```

```

        on UI.click b $ const $ do

```

```

          void $ element inputKey # set UI.value (unpack n)

```

```

          void $ element inputValue # set UI.value (show v)

```

```

          void $ element inputMap # set UI.value (show cmd)

```

```

        return b ]

```

```

    ]

```

```

    in UI.tr #. "itemrow" #+ entries

```

```

let apply2output f = do

```

```

  tgt ← getElementById window "output"

```

```

  case tgt of

```

```

    Nothing → pure ()

```

```

    Just t → f t

```

```

let listPairs cmd sel = do

```

```

  apply2output $ λt → void $ element t # set children [ ]

```

```

  cg ← liftIO $ readMVar (CM.getCG config)

```

```

  mapM_ (λ(n,v) → apply2output $ λt → void $ element t #+ [mkPairItem cmd n v]
    ) $ HM.toList (sel cg)

```

```

-- commands

```

```

let switchTo c@Backends = listPairs c CM.cgMapBackend

```

```

  switchTo c@Severities = listPairs c CM.cgMapSeverity

```

```

switchTo c@Scribes      = listPairs c CM.cgMapScribe
switchTo c@SubTrace     = listPairs c CM.cgMapSubtrace
switchTo c@Aggregation = listPairs c CM.cgMapAggregatedKind

let mkCommandButtons =
  let btns = map (\n → do
    b ← UI.button #. "w3-small w3-btn w3-ripple w3-grey" #+ [UI.bold #+ [string (show n)]]
    on UI.click b $ const $ (switchTo n)
    return b)
    [Backends, Scribes, Severities, SubTrace, Aggregation]
  in row btns

-- control global minimum severity
confMinSev ← liftIO $ minSeverity config
let setMinSev _el Nothing = pure ()
setMinSev _el (Just sev) = liftIO $ do
  setMinSeverity config (toEnum sev :: Severity)
  mkSevOption sev = UI.option # set UI.text (show sev)
  # set UI.value (show sev)
  # if (confMinSev == sev) then set UI.selected True else id
minsev ← UI.select #. "minsevfield" #+
  map mkSevOption (enumFrom Debug)-- for all severities
on UI.selectionChange minsev $ setMinSev minsev
let mkMinSevEntry = row [string "set minimum severity to:", UI.span # set html "&nbsp;&nbsp;  "; element]
let setError m = void $ element outputMsg # set UI.value ("ERROR: " ++ m)
let setMessage m = void $ element outputMsg # set UI.value m

-- construct row with input fields
let removeItem Backends k = CM.setBackends config k Nothing
removeItem Severities k = CM.setSeverity config k Nothing
removeItem Scribes k = CM.setScribes config k Nothing
removeItem SubTrace k = CM.setSubTrace config k Nothing
removeItem Aggregation k = CM.setAggregatedKind config k Nothing
let delItem = do
  k ← inputKey # get UI.value
  m ← inputMap # get UI.value
  case (readMay m :: Maybe Cmd) of
    Nothing → setError "parse error on cmd"
    Just c → do
      setMessage $ "deleting " ++ k ++ " from " ++ m
      liftIO $ removeItem c (pack k)
      switchTo c
let updateItem Backends k v = case (readMay v :: Maybe [BackendKind]) of
  Nothing → setError "parse error on backend list"
  Just v' → liftIO $ CM.setBackends config k $ Just v'
updateItem Severities k v = case (readMay v :: Maybe Severity) of
  Nothing → setError "parse error on severity"
  Just v' → liftIO $ CM.setSeverity config k $ Just v'
updateItem Scribes k v = case (readMay v :: Maybe [ScribeId]) of

```

```

    Nothing → setError "parse error on scribe list"
    Just v' → liftIO $ CM.setScribes config k $ Just v'
updateItem SubTrace k v = case (readMay v :: Maybe SubTrace) of
    Nothing → setError "parse error on subtrace"
    Just v' → liftIO $ CM.setSubTrace config k $ Just v'
updateItem Aggregation k v = case (readMay v :: Maybe AggregatedKind) of
    Nothing → setError "parse error on aggregated kind"
    Just v' → liftIO $ CM.setAggregatedKind config k $ Just v'

let setItem = do
    k ← inputKey # get UI.value
    v ← inputValue # get UI.value
    m ← inputMap # get UI.value
    case (readMay m :: Maybe Cmd) of
        Nothing → setError "parse error on cmd"
        Just c → do
            setMessage $ "setting " ++ k ++ " => " ++ v ++ " in " ++ m
            updateItem c (pack k) v
            switchTo c
let mkRowEdit = row [element inputKey, UI.span #. "w3-tag w3-round w3-blue midalign" # set UI.text "
mkRowBtns = row [do { b ← UI.button #. "w3-small w3-btn w3-ripple w3-teal" #+ [string "delete"]
                ; on UI.click b $ const $ (delItem)
                ; return b }
                , do { b ← UI.button #. "w3-small w3-btn w3-ripple w3-teal" #+ [string "store"]
                ; on UI.click b $ const $ (setItem)
                ; return b }
                ]
-- layout
let topGrid = UI.div #. "w3-panel" #+ [
    UI.div #. "w3-panel w3-border w3-border-blue" #+ [
        UI.div #. "w3-panel" #+ [mkMinSevEntry]
    ]
    , UI.div #. "w3-panel w3-border w3-border-blue" #+ [
        UI.div #. "w3-panel" #+ [UI.p # set UI.text "set or update a behaviour for a named log
        , UI.div #. "w3-panel" #+ [mkCommandButtons]
        , UI.div #. "w3-panel" #+ [mkRowEdit]
        , UI.div #. "w3-panel" #+ [mkRowBtns]
        , UI.div #. "w3-panel" #+ [element outputMsg]
    ]
]
tgt ← getElementById window "gridtarget"
case tgt of
    Nothing → pure ()
    Just t → void $ element t #+ [topGrid]

```

### 1.5.27 Cardano.BM.Output.Switchboard

#### Switchboard

We are using an *MVar* because we spawn a set of backends that may try to send messages to the switchboard before it is completely setup.

```

type SwitchboardMVar = MVar SwitchboardInternal
newtype Switchboard = Switchboard
  {getSB :: SwitchboardMVar}
data SwitchboardInternal = SwitchboardInternal
  {sbQueue :: TBQ.TBQueue NamedLogItem
  ,sbDispatch :: Async.Async ()
  }

```

#### Trace that forwards to the Switchboard

Every *Trace* ends in the *Switchboard* which then takes care of dispatching the messages to outputs

```

mainTrace :: Switchboard → TraceNamed IO
mainTrace sb = BaseTrace.BaseTrace $ Op $ λlognamed → do
  effectuate sb lognamed

```

#### Process incoming messages

Incoming messages are put into the queue, and then processed by the dispatcher. The queue is initialized and the message dispatcher launched.

```

instance IsEffectuator Switchboard where
  effectuate switchboard item = do
    let writequeue :: TBQ.TBQueue NamedLogItem → NamedLogItem → IO ()
        writequeue q i = do
          nocapacity ← atomically $ TBQ.isFullTBQueue q
          if nocapacity
            then handleOverflow switchboard
            else atomically $ TBQ.writeTBQueue q i
        sb ← readMVar (getSB switchboard)
        writequeue (sbQueue sb) item
    handleOverflow _ = putStrLn "Error: Switchboard's queue full, dropping log items!"

instead of 'writequeue ...':
  evalMonitoringAction config item ≫
    mapM_ (writequeue (sbQueue sb))
evalMonitoringAction :: Configuration → NamedLogItem → m [NamedLogItem]
evalMonitoringAction c item = return [item]
-- let action = LogNamed lnName=(lnName item) <> ".action", lnItem=LogMessage ...
-- return (action : item)

```

**Switchboard** implements **Backend** functions**Switchboard** is an **Declaration of a Backend**

```

instance IsBackend Switchboard where
  typeof _ = SwitchboardBK
  realize cfg =
    let spawnDispatcher
      :: Configuration
      → [(BackendKind, Backend)]
      → TBQ.TBQueue NamedLogItem
      → IO (Async.Async ())
    spawnDispatcher config backends queue = do
      now ← getCurrentTime
      let messageCounters = resetCounters now
      countersMVar ← newMVar messageCounters
      let traceInQueue q =
        BaseTrace.BaseTrace $ Op $ \lognamed → do
          nocapacity ← atomically $ TBQ.isFullTBQueue q
          if nocapacity
          then putStrLn "Error: Switchboard's queue full, dropping log items!"
          else atomically $ TBQ.writeTBQueue q lognamed
      ctx = TraceContext { loggerName = ""
        , configuration = cfg
        , minSeverity = Debug
        , tracetype = Neutral
        }
      _timer ← Async.async $ sendAndResetAfter
        (ctx, traceInQueue queue)
        "#messagecounters.switchboard"
      countersMVar
      60000 -- 60000 ms = 1 min
      Warning -- Debug
    let sendMessage nli befilter = do
      selectedBackends ← getBackends config (lnName nli)
      let selBEs = befilter selectedBackends
      forM_ backends $ \ (bek, be) →
        when (bek ∈ selBEs) (bEffectuate be $ nli)
    qProc counters = do
      nli ← atomically $ TBQ.readTBQueue queue
      -- do not count again messages that contain the results of message counters
      when (lnName nli ≠ "#messagecounters.switchboard") $
        -- increase the counter for the specific severity
        modifyMVar_ counters $ \cnt → return $ updateMessageCounters cnt $ lnItem nli
    case lnItem nli of
      LogObject _ KillPill →
        forM_ backends (λ(–, be) → bUnrealize be)
  # ifdef ENABLE_AGGREGATION

```

```

    LogObject _ (AggregatedMessage _) → do
      sendMessage nli (filter (≠ AggregationBK))
      qProc counters
# endif
# ifdef ENABLE_MONITORING
    LogObject _ (MonitoringEffect inner) → do
      sendMessage (nli {lnItem = inner}) (filter (≠ MonitoringBK))
      qProc counters
# endif
    _ → sendMessage nli id >> qProc counters

  Async.async $ qProc countersMVar
in do
    q ← atomically $ TBQ.newTBQueue 2048
    sbref ← newEmptyMVar
    let sb :: Switchboard = Switchboard sbref
    backends ← getSetupBackends cfg
    bs ← setupBackends backends cfg sb []
    dispatcher ← spawnDispatcher cfg bs q
    -- link the given Async to the current thread, such that if the Async
    -- raises an exception, that exception will be re-thrown in the current
    -- thread, wrapped in ExceptionInLinkedThread.
    Async.link dispatcher
    putMVar sbref $ SwitchboardInternal {sbQueue = q, sbDispatch = dispatcher}
    return sb

  unrealize switchboard = do
    let clearMVar :: MVar a → IO ()
      clearMVar = void ∘ tryTakeMVar
    (dispatcher, queue) ← withMVar (getSB switchboard) (λsb → return (sbDispatch sb, sbQueue sb))
    -- send terminating item to the queue
    lo ← LogObject <$> (mkLOMeta Warning) <*> pure KillPill
    atomically $ TBQ.writeTBQueue queue $ LogNamed "kill.switchboard" lo
    -- wait for the dispatcher to exit
    res ← Async.waitCatch dispatcher
    either throwM return res
    (clearMVar ∘ getSB) switchboard

```

### Realizing the backends according to configuration

```

setupBackends :: [BackendKind]
  → Configuration
  → Switchboard
  → [(BackendKind, Backend)]
  → IO [(BackendKind, Backend)]
setupBackends [] _ _ acc = return acc
setupBackends (bk : bes) c sb acc = do
  be' ← setupBackend' bk c sb

```

```

    setupBackends bes c sb ((bk, be') : acc)
  setupBackend' :: BackendKind → Configuration → Switchboard → IO Backend
  setupBackend' SwitchboardBK _ _ = error "cannot instantiate a further Switchboard"
  # ifdef ENABLE_MONITORING
  setupBackend' MonitoringBK c sb = do
    let trace = mainTrace sb
    ctx = TraceContext { loggerName = ""
                        , configuration = c
                        , minSeverity = Debug
                        , tracetype = Neutral
                        }
    be :: Cardano.BM.Output ◦ Monitoring.Monitor ← Cardano.BM.Output ◦ Monitoring.realizefrom (ctx, trace) sb
    return MkBackend
      { bEffectuate = Cardano.BM.Output ◦ Monitoring.effectuate be
      , bUnrealize = Cardano.BM.Output ◦ Monitoring.unrealize be
      }
  # else
  -- We need it anyway, to avoid "Non-exhaustive patterns" warning.
  setupBackend' MonitoringBK _ _ =
    error "Impossible happened: monitoring is disabled by Cabal-flag, we mustn't match this backend"
  # endif
  # ifdef ENABLE_EKG
  setupBackend' EKGViewBK c sb = do
    let trace = mainTrace sb
    ctx = TraceContext { loggerName = ""
                        , configuration = c
                        , minSeverity = Debug
                        , tracetype = Neutral
                        }
    be :: Cardano.BM.Output ◦ EKGView.EKGView ← Cardano.BM.Output ◦ EKGView.realizefrom (ctx, trace) sb
    return MkBackend
      { bEffectuate = Cardano.BM.Output ◦ EKGView.effectuate be
      , bUnrealize = Cardano.BM.Output ◦ EKGView.unrealize be
      }
  # else
  -- We need it anyway, to avoid "Non-exhaustive patterns" warning.
  setupBackend' EKGViewBK _ _ =
    error "Impossible happened: EKG is disabled by Cabal-flag, we mustn't match this backend"
  # endif
  # ifdef ENABLE_AGGREGATION
  setupBackend' AggregationBK c sb = do
    let trace = mainTrace sb
    ctx = TraceContext { loggerName = ""
                        , configuration = c
                        , minSeverity = Debug
                        , tracetype = Neutral
                        }

```

```

be :: Cardano.BM.Output ◦ Aggregation.Aggregation ← Cardano.BM.Output ◦ Aggregation.realizefrom (ctx,
return MkBackend
  { bEffectuate = Cardano.BM.Output ◦ Aggregation.effectuate be
  , bUnrealize = Cardano.BM.Output ◦ Aggregation.unrealize be
  }
# else
-- We need it anyway, to avoid "Non-exhaustive patterns" warning.
setupBackend' AggregationBK _ =
  error "Impossible happened: aggregation is disabled by Cabal-flag, we mustn't match this"
# endif
setupBackend' KatipBK c _ = do
  be :: Cardano.BM.Output ◦ Log.Log ← Cardano.BM.Output ◦ Log.realize c
  return MkBackend
    { bEffectuate = Cardano.BM.Output ◦ Log.effectuate be
    , bUnrealize = Cardano.BM.Output ◦ Log.unrealize be
    }

```

### 1.5.28 Cardano.BM.Output.Log

#### Internal representation

```

type LogMVar = MVar LogInternal
newtype Log = Log
  { getK :: LogMVar }
data LogInternal = LogInternal
  { kLogEnv      :: K.LogEnv
  , msgCounters :: MessageCounter
  , configuration :: Config.Configuration }

```

#### Log implements effectuate

```

instance IsEffectuator Log where
  effectuate katip item = do
    let logMVar = getK katip
    c ← configuration < $ > readMVar logMVar
    setupScribes ← getSetupScribes c
    selscribes ← getScribes c (lnName item)
    let selscribesFiltered =
      case lnItem item of
        LogObject _ (LogMessage (LogItem Private _))
          → removePublicScribes setupScribes selscribes
        _ → selscribes
    forM_ selscribesFiltered $ λsc → passN sc katip item
    -- increase the counter for the specific severity and message type
    modifyMVar_ logMVar $ λli → return $
      li { msgCounters = updateMessageCounters (msgCounters li) (lnItem item) }

```



```

-- reset message counters after 60 sec = 1 min
resetMessageCounters logMVar 60 Warning selscribesFiltered
where
removePublicScribes allScribes = filter $ \sc →
  let (_, nameD) = T.breakOn " :: " sc
    -- drop " :: " from the start of name
    name = T.drop 2 nameD
  in
  case find (λx → scName x ≡ name) allScribes of
    Nothing → False
    Just scribe → scPrivacy scribe ≡ ScPrivate
resetMessageCounters logMVar interval sev scribes = do
  counters ← msgCounters < $ > readMVar logMVar
  let start = mcStart counters
    now = case lItem item of
      LogObject meta _ → tstamp meta
    diffTime = round $ diffUTCTime now start
  when (diffTime > interval) $ do
    countersObjects ← forM (HM.toList $ mcCountersMap counters) $ λ(key, count) →
      LogObject
        < $ > (mkLOMeta sev)
        < * > pure (LogValue (pack key) (PureI $ toInteger count))
    intervalObject ←
      LogObject
        < $ > (mkLOMeta sev)
        < * > pure (LogValue "time_interval_(s)" (PureI diffTime))
  let namedCounters = map (λlo → LogNamed "#messagecounters.katip" lo)
    (countersObjects ++ [intervalObject])
  forM_ scribes $ \sc →
    forM_ namedCounters $ λnamedCounter →
      passN sc katip namedCounter
  modifyMVar_ logMVar $ λli → return $
    li {msgCounters = resetCounters now}
handleOverflow _ = putStrLn "Notice: Katip's queue full, dropping log items!"

```

### Log implements backend functions

```

instance IsBackend Log where
  typeof _ = KatipBK
  realize config = do
    let updateEnv :: K.LogEnv → IO UTCTime → K.LogEnv
      updateEnv le timer =
        le {K._logEnvTimer = timer, K._logEnvHost = "hostname"}
    register :: [ScribeDefinition] → K.LogEnv → IO K.LogEnv
    register [ ] le = return le
    register (defsc : dscs) le = do

```

```

let kind = scKind defsc
    name = scName defsc
    rotParams = scRotation defsc
    name' = pack (show kind) <> ":" <> name
scr ← createScribe kind name rotParams
register dcs ≡ K.registerScribe name' scr scribeSettings le
mockVersion :: Version
mockVersion = Version [0,1,0,0] []
scribeSettings :: KC.ScribeSettings
scribeSettings =
    let bufferSize = 5000 -- size of the queue (in log items)
    in
        KC.ScribeSettings bufferSize
createScribe FileTextSK name rotParams = mkTextFileScribe
    rotParams
    (FileDescription $ unpack name)
    False
createScribe FileJsonSK name rotParams = mkJsonFileScribe
    rotParams
    (FileDescription $ unpack name)
    False
createScribe StdoutSK _ _ = mkStdoutScribe
createScribe StderrSK _ _ = mkStderrScribe
cfoKey ← Config.getOptionOrDefault config (pack "cfokey") (pack "<unknown>")
le0 ← K.initLogEnv
    (K.Namespace [ "iohk" ])
    (fromString $ (unpack cfoKey) <> ":" <> showVersion mockVersion)
-- request a new time 'getCurrentTime' at most 100 times a second
timer ← mkAutoUpdate defaultUpdateSettings {updateAction = getCurrentTime, updateFreq = 10000}
let le1 = updateEnv le0 timer
scribes ← getSetupScribes config
le ← register scribes le1
messageCounters ← resetCounters < $ > getCurrentTime
kref ← newMVar $ LogInternal le messageCounters config
return $ Log kref
unrealize katip = do
    le ← withMVar (getK katip) $ λk → return (kLogEnv k)
    void $ K.closeScribes le

```

```

example :: IO ()
example = do
    config ← Config.setup "from_some_path.yaml"
    k ← setup config
    passN (pack (show StdoutSK)) k $ LogNamed
        { lnName = "test"
        , lnItem = LogMessage $ LogItem

```

```

    {liSelection = Both
    ,liSeverity = Info
    ,liPayload = "Hello!"
    }
  }
passN (pack (show StdoutSK)) k $ LogNamed
  {lnName = "test"
  ,lnItem = LogValue "cpu-no" 1
  }

```

Needed instances for *katip*:

```

deriving instance K.ToObject LogObject
deriving instance K.ToObject LogItem
deriving instance K.ToObject (Maybe LOContent)
instance KC.LogItem LogObject where
  payloadKeys _ = KC.AllKeys
instance KC.LogItem LogItem where
  payloadKeys _ = KC.AllKeys
instance KC.LogItem (Maybe LOContent) where
  payloadKeys _ = KC.AllKeys

```

### Log.passN

The following function copies the **NamedLogItem** to the queues of all scribes that match on their name. Compare start of name of scribe to (*show backend* <> " : "). This function is non-blocking.

```

passN :: ScribeId → Log → NamedLogItem → IO ()
passN backend katip namedLogItem = do
  env ← kLogEnv < $ > readMVar (getK katip)
  forM_ (Map.toList $ K._logEnvScribes env) $
    λ(scName, (KC.ScribeHandle _ shChan)) →
      -- check start of name to match ScribeKind
      if backend `isPrefixOf` scName
      then do
        let (LogObject lometa loitem) = lnItem namedLogItem
        let (sev, msg, payload) = case loitem of
          (LogMessage logItem) →
            (severity lometa, liPayload logItem, Nothing)
          (ObserveDiff _) →
            let text = TL.toStrict (encodeToLazyText loitem)
            in
              (severity lometa, text, Just loitem)
          (ObserveOpen _) →
            let text = TL.toStrict (encodeToLazyText loitem)
            in
              (severity lometa, text, Just loitem)
          (ObserveClose _) →

```

```

    let text = TL.toStrict (encodeToLazyText loitem)
    in
      (severity lometa, text, Just loitem)
  (AggregatedMessage aggregated) →
    let text = T.concat $ (flip map) aggregated $ \ (name, agg) →
      "\n" <> name <> ": " <> pack (show agg)
    in
      (severity lometa, text, Nothing)
  (LogValue name value) →
    (severity lometa, name <> " = " <> pack (showSI value), Nothing)
  (MonitoringEffect logitem) →
    let text = TL.toStrict (encodeToLazyText logitem)
    in
      (severity lometa, text, Just loitem)
  KillPill →
    (severity lometa, "Kill pill received!", Nothing)
if (msg ≡ "") ∧ (isNothing payload)
then return ()
else do
  let threadIdText = KC.ThreadIdText $ tid lometa
  let ns = lnName namedLogItem
  let itemTime = tstamp lometa
  let itemKatip = K.Item {
    _itemApp      = env ^. KC.logEnvApp
  , _itemEnv      = env ^. KC.logEnvEnv
  , _itemSeverity = sev2klog sev
  , _itemThread   = threadIdText
  , _itemHost     = env ^. KC.logEnvHost
  , _itemProcess  = env ^. KC.logEnvPid
  , _itemPayload  = payload
  , _itemMessage  = K.logStr msg
  , _itemTime     = itemTime
  , _itemNamespace = (env ^. KC.logEnvApp) <> (K.Namespace [ ns ])
  , _itemLoc      = Nothing
  }
  void $ atomically $ KC.tryWriteTBQueue shChan (KC.NewItem itemKatip)
else return ()

```

## Scribes

```

mkStdoutScribe :: IO K.Scribe
mkStdoutScribe = do
  -- duplicate stdout so that Katip's closing
  -- action will not close the real stdout
  stdout' ← hDuplicate stdout
  mkTextFileScribeH stdout' True
mkStderrScribe :: IO K.Scribe

```

```

mkStderrScribe = do
  -- duplicate stderr so that Katip's closing
  -- action will not close the real stderr
  stderr' ← hDuplicate stderr
  mkTextFileScribeH stderr' True

mkTextFileScribeH :: Handle → Bool → IO K.Scribe
mkTextFileScribeH handler color = do
  mkFileScribeH handler formatter color
  where
    formatter h colorize verbosity item =
      TIO.hPutStrLn h $! toLazyText $ formatItem colorize verbosity item

mkFileScribeH
  :: Handle
  → (forall a ◦ K.LogItem a ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ())
  → Bool
  → IO K.Scribe
mkFileScribeH h formatter colorize = do
  hSetBuffering h LineBuffering
  locklocal ← newMVar ()
  let logger :: forall a ◦ K.LogItem a ⇒ K.Item a → IO ()
      logger item = withMVar locklocal $ \_ →
        formatter h colorize K.V0 item
  pure $ K.Scribe logger (hClose h)

mkTextFileScribe :: Maybe RotationParameters → FileDescription → Bool → IO K.Scribe
mkTextFileScribe rotParams fdesc colorize = do
  mkFileScribe rotParams fdesc formatter colorize
  where
    formatter :: Handle → Bool → K.Verbosity → K.Item a → IO Int
    formatter hdl colorize' v' item =
      case KC._itemMessage item of
        K.LogStr "" →
          -- if message is empty do not output it
          return 0
        _ → do
          let tmsg = toLazyText $ formatItem colorize' v' item
          TIO.hPutStrLn hdl tmsg
          return $ fromIntegral $ TL.length tmsg

mkJsonFileScribe :: Maybe RotationParameters → FileDescription → Bool → IO K.Scribe
mkJsonFileScribe rotParams fdesc colorize = do
  mkFileScribe rotParams fdesc formatter colorize
  where
    formatter :: (K.LogItem a) ⇒ Handle → Bool → K.Verbosity → K.Item a → IO Int
    formatter h _ verbosity item = do
      let jmsg = case KC._itemMessage item of
        -- if a message is contained in item then only the
        -- message is printed and not the data
        K.LogStr "" → K.itemJson verbosity item

```

```

    K.LogStr msg → K.itemJson verbosity $
      item {KC._itemMessage = K.logStr (" " :: Text)
        , KC._itemPayload = LogItem Both $ TL.toStrict $ toLazyText msg
        -- do we need the severity from meta?
        }
      tmsg = encodeToLazyText jmsg
      TIO.hPutStrLn h tmsg
      return $ fromIntegral $ TL.length tmsg
mkFileScribe
  :: Maybe RotationParameters
  → FileDescription
  → (forall a ◦ K.LogItem a ⇒ Handle → Bool → K.Verbosity → K.Item a → IO Int)
  → Bool
  → IO K.Scribe
mkFileScribe (Just rotParams) fdesc formatter colorize = do
  let prefixDir = prefixPath fdesc
  (createDirectoryIfMissing True prefixDir)
  'catchIO' (prtoutException ("cannot log prefix directory: " ++ prefixDir))
  let fpath = filePath fdesc
  trp ← initializeRotator rotParams fpath
  scribestate ← newMVar trp -- triple of (handle), (bytes remaining), (rotate time)
  -- sporadically remove old log files - every 10 seconds
  cleanup ← mkAutoUpdate defaultUpdateSettings {
    updateAction = cleanupRotator rotParams fpath
    , updateFreq = 10000000
  }
  let finalizer :: IO ()
      finalizer = withMVar scribestate $
        λ(h, -, -) → hClose h
  let logger :: forall a ◦ K.LogItem a ⇒ K.Item a → IO ()
      logger item =
        modifyMVar_ scribestate $ λ(h, bytes, rottime) → do
          byteswritten ← formatter h colorize K.V0 item
          -- remove old files
          cleanup
          -- detect log file rotation
          let bytes' = bytes - (toInteger $ byteswritten)
          let tdiff' = round $ diffUTCTime rottime (K._itemTime item)
          if bytes' < 0 ∨ tdiff' < (0 :: Integer)
          then do -- log file rotation
            hClose h
            (h2, bytes2, rottime2) ← evalRotator rotParams fpath
            return (h2, bytes2, rottime2)
          else
            return (h, bytes', rottime)
  return $ K.Scribe logger finalizer
-- log rotation disabled.

```

```

mkFileScribe Nothing fdesc formatter colorize = do
  let prefixDir = prefixPath fdesc
  (createDirectoryIfMissing True prefixDir)
  'catchIO' (prtoutException ("cannot log prefix directory: " ++ prefixDir))
  let fpath = filePath fdesc
  h ← catchIO (openFile fpath WriteMode) $
    λe → do
      prtoutException ("error while opening log: " ++ fpath) e
      -- fallback to standard output in case of exception
      return stdout
  hSetBuffering h LineBuffering
  scribestate ← newMVar h
  let finalizer :: IO ()
      finalizer = withMVar scribestate hClose
  let logger :: forall a . K.LogItem a ⇒ K.Item a → IO ()
      logger item =
        withMVar scribestate $ λhandler →
          void $ formatter handler colorize K.V0 item
  return $ K.Scribe logger finalizer

formatItem :: Bool → K.Verbosity → K.Item a → Builder
formatItem withColor _verb K.Item {..} =
  fromText header <>
  fromText " " <>
  brackets (fromText timestamp) <>
  fromText " " <>
  KC.unLogStr _itemMessage
where
  header = colorBySeverity _itemSeverity $
    "[" <> mconcat [namedcontext <> ": " <> severity <> ": " <> threadid <> "]"
  namedcontext = KC.intercalateNs _itemNamespace
  severity = KC.renderSeverity _itemSeverity
  threadid = KC.getThreadIdText _itemThread
  timestamp = pack $ formatTime defaultTimeLocale tsformat _itemTime
  tsformat :: String
  tsformat = "%F %T%2Q %Z"
  colorBySeverity s m = case s of
    K.EmergencyS → red m
    K.AlertS     → red m
    K.CriticalS  → red m
    K.ErrorS     → red m
    K.NoticeS    → magenta m
    K.WarningS   → yellow m
    K.InfoS      → blue m
    _            → m
  red = colorize "31"
  yellow = colorize "33"
  magenta = colorize "35"

```

```

    blue = colorize "34"
    colorize c m
      | withColor = "\ESC[" <> c <> "m" <> m <> "\ESC[0m"
      | otherwise = m
-- translate Severity to Log.Severity
sev2klog :: Severity → K.Severity
sev2klog = λcase
  Debug    → K.DebugS
  Info     → K.InfoS
  Notice   → K.NoticeS
  Warning  → K.WarningS
  Error    → K.ErrorS
  Critical → K.CriticalS
  Alert    → K.AlertS
  Emergency → K.EmergencyS

data FileDescription = FileDescription {
  filePath :: !FilePath
  deriving (Show)
  prefixPath :: FileDescription → FilePath
  prefixPath = takeDirectory ∘ filePath

```

### 1.5.29 Cardano.BM.Output.EKGView

#### Structure of EKGView

```

type EKGViewMVar = MVar EKGViewInternal
newtype EKGView = EKGView
  {getEV :: EKGViewMVar}
data EKGViewInternal = EKGViewInternal
  {evQueue :: TBQ.TBQueue (Maybe NamedLogItem)
  ,evLabels :: EKGViewMap
  ,evServer :: Server
  }

```

#### Relation from variable name to label handler

We keep the label handlers for later update in a *HashMap*.

```

type EKGViewMap = HM.HashMap Text Label.Label

```

#### Internal Trace

This is an internal **Trace**, named "#ekgview", which can be used to control the messages that are being displayed by EKG.



```

ekgTrace :: EKGView → Configuration → IO (Trace IO)
ekgTrace ekg c = do
  let trace = ekgTrace' ekg
      ctx = TraceContext { loggerName = " "
                          , configuration = c
                          , minSeverity = Debug
                          , tracetype = Neutral
                          }
      Trace.subTrace "#ekgview" (ctx, trace)
  where
    ekgTrace' :: EKGView → TraceNamed IO
    ekgTrace' ekgview = BaseTrace.BaseTrace $ Op $ λ(LogNamed lognamed lo) → do
      let setlabel :: Text → Text → EKGViewInternal → IO (Maybe EKGViewInternal)
          setlabel name label ekg_i@(EKGViewInternal _ labels server) =
            case HM.lookup name labels of
              Nothing → do
                ekghdl ← getLabel name server
                Label.set ekghdl label
                return $ Just $ ekg_i { evLabels = HM.insert name ekghdl labels }
              Just ekghdl → do
                Label.set ekghdl label
                return Nothing
      update :: LogObject → LoggerName → EKGViewInternal → IO (Maybe EKGViewInternal)
      update (LogObject _ (LogMessage logitem)) logname ekg_i =
        setlabel logname (liPayload logitem) ekg_i
      update (LogObject _ (LogValue iname value)) logname ekg_i =
        let logname' = logname <> "." <> iname
        in
          setlabel logname' (pack $ show value) ekg_i
      update _ _ _ = return Nothing
    modifyMVar_ (getEV ekgview) $ λekgup → do
      let -- strip off some prefixes not necessary for display
          lognam1 = case stripPrefix "#ekgview.#aggregation." lognamed of
            Nothing → lognamed
            Just ln' → ln'
          logname = case stripPrefix "#ekgview." lognam1 of
            Nothing → lognam1
            Just ln' → ln'
      upd ← update lo logname ekgup
      case upd of
        Nothing → return ekgup
        Just ekgup' → return ekgup'

```

### EKG view is an effectuator

Function *effectuate* is called to pass in a **NamedLogItem** for display in EKG. If the log item is an *AggregatedStats* message, then all its constituents are put into the queue. In case the queue is full,

all new items are dropped.

```

instance IsEffectuator EKGView where
  effectuate ekgview item = do
    ekg ← readMVar (getEV ekgview)
    let enqueue a = do
      nocapacity ← atomically $ TBQ.isFullTBQueue (evQueue ekg)
      if nocapacity
      then handleOverflow ekgview
      else atomically $ TBQ.writeTBQueue (evQueue ekg) (Just a)
    case (lnItem item) of
      (LogObject lometa (AggregatedMessage ags)) → liftIO $ do
        let logname = lnName item
        traceAgg :: [(Text, Aggregated)] → IO ()
        traceAgg [] = return ()
        traceAgg ((n, AggregatedEWMA ewma) : r) = do
          enqueue $ LogNamed (logname <> "." <> n) $ LogObject lometa (LogValue "avg" $ avg ewma)
          traceAgg r
        traceAgg ((n, AggregatedStats stats) : r) = do
          let statsname = logname <> "." <> n
          qbasestats s' nm = do
            enqueue $ LogNamed nm $ LogObject lometa (LogValue "mean" (PureD $ meanOfStats s'))
            enqueue $ LogNamed nm $ LogObject lometa (LogValue "min" $ fmin s')
            enqueue $ LogNamed nm $ LogObject lometa (LogValue "max" $ fmax s')
            enqueue $ LogNamed nm $ LogObject lometa (LogValue "count" $ PureI $ fromIntegral $ fcount s')
            enqueue $ LogNamed nm $ LogObject lometa (LogValue "stdev" (PureD $ stdevOfStats s'))
          enqueue $ LogNamed statsname $ LogObject lometa (LogValue "last" $ flast stats)
          qbasestats (fbasic stats) $ statsname <> ".basic"
          qbasestats (fdelta stats) $ statsname <> ".delta"
          qbasestats (ftimed stats) $ statsname <> ".timed"
          traceAgg r
        traceAgg ags
      (LogObject _ (LogMessage _)) → enqueue item
      (LogObject _ (LogValue _)) → enqueue item
      _ → return ()
  handleOverflow _ = putStrLn "Notice: EKGViews's queue full, dropping log items!"

```

**EKGView** implements **Backend** functions

**EKGView** is an **IsBackend**

```

instance IsBackend EKGView where
  typeof _ = EKGViewBK
  realize _ = error "EKGView cannot be instantiated by 'realize'"
  realizefrom sbtrace@(ctx, _) _ = do
    let config = configuration ctx
    evref ← newEmptyMVar

```

```

let ekgview = EKGView evref
evport ← getEKGport config
ehdl ← forkServer "127.0.0.1" evport
ekghdl ← getLabel "iohk-monitoring version" ehdl
Label.set ekghdl $ pack (showVersion version)
ekgtrace ← ekgTrace ekgview config
queue ← atomically $ TBQ.newTBQueue 512
dispatcher ← spawnDispatcher queue sbtrace ekgtrace
-- link the given Async to the current thread, such that if the Async
-- raises an exception, that exception will be re-thrown in the current
-- thread, wrapped in ExceptionInLinkedThread.
Async.link dispatcher
putMVar evref $ EKGViewInternal
  {evLabels = HM.empty
  ,evServer = ehdl
  ,evQueue = queue
  }
return ekgview
unrealize ekgview =
  withMVar (getEV ekgview) $ \ekg →
    killThread $ serverThreadId $ evServer ekg

```

### Asynchronously reading log items from the queue and their processing

```

spawnDispatcher :: TBQ.TBQueue (Maybe NamedLogItem)
  → Trace.Trace IO
  → Trace.Trace IO
  → IO (Async.Async ())
spawnDispatcher evqueue sbtrace trace = do
  now ← getCurrentTime
  let messageCounters = resetCounters now
  countersMVar ← newMVar messageCounters
  _timer ← Async.async $ sendAndResetAfter
    sbtrace
    "#messagecounters.ekgview"
    countersMVar
    60000 -- 60000 ms = 1 min
    Warning -- Debug
  Async.async $ qProc countersMVar
where
  qProc counters = do
    maybeItem ← atomically $ TBQ.readTBQueue evqueue
    case maybeItem of
      Just (LogNamed logname logvalue@(LogObject _ _)) → do
        trace' ← Trace.appendName logname trace
        Trace.traceConditionally trace' logvalue

```

```

-- increase the counter for the type of message
modifyMVar_counters $ \cnt → return $ updateMessageCounters cnt logvalue
qProc counters
Nothing → return () -- stop here

```

### Interactive testing **EKGView**

```

test :: IO ()
test = do
  c ← Cardano.BM.Setup.setupTrace (Left "test/config.yaml") "ekg"
  ev ← Cardano.BM.Output ◦ EKGView.realize c
  effectuate ev $ LogNamed "test.questions" (LogValue "answer" 42)
  effectuate ev $ LogNamed "test.monitor023" (LogMessage (LogItem Public Warning "!!!! ALARM !!!!")

```

### 1.5.30 Cardano.BM.Output.Aggregation

#### Internal representation

```

type AggregationMVar = MVar AggregationInternal
newtype Aggregation = Aggregation
  {getAg :: AggregationMVar}
data AggregationInternal = AggregationInternal
  {agQueue :: TBQ.TBQueue (Maybe NamedLogItem)
  ,agDispatch :: Async.Async ()
  }

```

#### Relation from context name to aggregated statistics

We keep the aggregated values (**Aggregated**) for a named context in a *HashMap*.

```

type AggregationMap = HM.HashMap Text AggregatedExpanded

```

#### Info for Aggregated operations

Apart from the **Aggregated** we keep some valuable info regarding to them; such as when was the last time it was sent.

```

type Timestamp = Word64
data AggregatedExpanded = AggregatedExpanded
  {aeAggregated :: Aggregated
  ,aeResetAfter :: !(Maybe Int)
  ,aeLastSent :: {-# UNPACK #-} !Timestamp
  }

```

**Aggregation** implements *effectuate*

**Aggregation** is an **IsEffectuator** Enter the log item into the **Aggregation** queue.

```
instance IsEffectuator Aggregation where
  effectuate agg item = do
    ag ← readMVar (getAg agg)
    nocapacity ← atomically $ TBQ.isFullTBQueue (agQueue ag)
    if nocapacity
    then handleOverflow agg
    else atomically $ TBQ.writeTBQueue (agQueue ag) $! Just item
  handleOverflow _ = putStrLn "Notice: Aggregation's queue full, dropping log items!"
```

**Aggregation** implements **Backend** functions

**Aggregation** is an **IsBackend**

```
instance IsBackend Aggregation where
  typeOf _ = AggregationBK
  realize _ = error "Aggregation cannot be instantiated by 'realize'"
  realizeFrom trace0@(ctx, _) _ = do
    trace ← Trace.subTrace "#aggregation" trace0
    aggref ← newEmptyMVar
    aggregationQueue ← atomically $ TBQ.newTBQueue 2048
    dispatcher ← spawnDispatcher (configuration ctx) HM.empty aggregationQueue trace
    -- link the given Async to the current thread, such that if the Async
    -- raises an exception, that exception will be re-thrown in the current
    -- thread, wrapped in ExceptionInLinkedThread.
    Async.link dispatcher
    putMVar aggref $ AggregationInternal aggregationQueue dispatcher
    return $ Aggregation aggref
  unrealize aggregation = do
    let clearMVar :: MVar a → IO ()
        clearMVar = void ∘ tryTakeMVar
    (dispatcher, queue) ← withMVar (getAg aggregation) (λag →
      return (agDispatch ag, agQueue ag))
    -- send terminating item to the queue
    atomically $ TBQ.writeTBQueue queue Nothing
    -- wait for the dispatcher to exit
    -- TODO add a timeout to waitCatch in order
    -- to be sure that it will finish
    res ← Async.waitCatch dispatcher
    either throwM return res
    (clearMVar ∘ getAg) aggregation
```

## Asynchronously reading log items from the queue and their processing

```

spawnDispatcher :: Configuration
    → AggregationMap
    → TBQ.TBQueue (Maybe NamedLogItem)
    → Trace.Trace IO
    → IO (Async.Async ())
spawnDispatcher conf aggMap aggregationQueue trace = do
    now ← getCurrentTime
    let messageCounters = resetCounters now
    countersMVar ← newMVar messageCounters
    _timer ← Async.async $ sendAndResetAfter
        trace
        "#messagecounters.aggregation"
        countersMVar
        60000 -- 60000 ms = 1 min
        Warning -- Debug
    Async.async $ qProc countersMVar aggMap
where
    qProc counters aggregatedMap = do
        maybeItem ← atomically $ TBQ.readTBQueue aggregationQueue
        case maybeItem of
            Just (LogNamed logname lo@(LogObject lm _)) → do
                (updatedMap, aggregations) ← update lo logname aggregatedMap
                unless (null aggregations) $
                    sendAggregated (LogObject lm (AggregatedMessage aggregations)) logname
                -- increase the counter for the specific severity and message type
                modifyMVar_counters $ \cnt → return $ updateMessageCounters cnt lo
                qProc counters updatedMap
            Nothing → return ()
    createNupdate name value lme agmap = do
        case HM.lookup name agmap of
            Nothing → do
                -- if Aggregated does not exist; initialize it.
                aggregatedKind ← getAggregatedKind conf name
                case aggregatedKind of
                    StatsAK → return $ singletonStats value
                    EwmaAK aEWMA → do
                        let initEWMA = EmptyEWMA aEWMA
                        return $ AggregatedEWMA $ ewma initEWMA value
                Just a → return $ updateAggregation value (aeAggregated a) lme (aeResetAfter a)
    update :: LogObject
        → LoggerName
        → AggregationMap
        → IO (AggregationMap, [(Text, Aggregated)])
    update (LogObject lme (LogValue iname value)) logname agmap = do
        let fullname = logname <> " ." <> iname

```

```

    aggregated ← createNupdate fullname value lme agmap
    now ← getMonotonicTimeNSec
    let aggregatedX = AggregatedExpanded {
        aeAggregated = aggregated
        ,aeResetAfter = Nothing
        ,aeLastSent = now
    }
    namedAggregated = [(iname,aeAggregated aggregatedX)]
    updatedMap = HM.alter (const $ Just $ aggregatedX) fullname agmap
    return (updatedMap,namedAggregated)

update (LogObject lme (ObserveDiff counterState)) logname agmap =
    updateCounters (csCounters counterState) lme (logname,"diff") agmap []
update (LogObject lme (ObserveOpen counterState)) logname agmap =
    updateCounters (csCounters counterState) lme (logname,"open") agmap []
update (LogObject lme (ObserveClose counterState)) logname agmap =
    updateCounters (csCounters counterState) lme (logname,"close") agmap []
update (LogObject lme (LogMessage _)) logname agmap = do
    let iname = pack $ show (severity lme)
    let fullname = logname <> "." <> iname
    aggregated ← createNupdate fullname (PureI 0) lme agmap
    now ← getMonotonicTimeNSec
    let aggregatedX = AggregatedExpanded {
        aeAggregated = aggregated
        ,aeResetAfter = Nothing
        ,aeLastSent = now
    }
    namedAggregated = [(iname,aeAggregated aggregatedX)]
    updatedMap = HM.alter (const $ Just $ aggregatedX) fullname agmap
    return (updatedMap,namedAggregated)

-- everything else
update _ _ agmap = return (agmap,[])
updateCounters :: [Counter]
    → LOMeta
    → (LoggerName,LoggerName)
    → AggregationMap
    → [(Text,Aggregated)]
    → IO (AggregationMap,[(Text,Aggregated)])
updateCounters [] _ _ aggrMap aggs = return $ (aggrMap,aggs)
updateCounters (counter : cs) lme (logname,msgname) aggrMap aggs = do
    let name = cName counter
        subname = msgname <> "." <> (nameCounter counter) <> "." <> name
        fullname = logname <> "." <> subname
        value = cValue counter
    aggregated ← createNupdate fullname value lme aggrMap
    now ← getMonotonicTimeNSec
    let aggregatedX = AggregatedExpanded {
        aeAggregated = aggregated

```

```

    ,aeResetAfter = Nothing
    ,aeLastSent = now
  }
  namedAggregated = (subname,aggregated)
  updatedMap = HM.alter (const $ Just $ aggregatedX) fullname aggrMap
  updateCounters cs lme (logname,msgname) updatedMap (namedAggregated:aggs)
sendAggregated :: LogObject → Text → IO ()
sendAggregated aggregatedMsg@(LogObject _ (AggregatedMessage _)) logname = do
  -- enter the aggregated message into the Trace
  trace' ← Trace.appendName logname trace
  liftIO $ Trace.traceConditionally trace' aggregatedMsg
-- ignore every other message
sendAggregated _ _ = return ()

```

## Update aggregation

We distinguish an uninitialized from an already initialized aggregation. The latter is properly initialized.

We use Welford's online algorithm to update the estimation of mean and variance of the sample statistics. (see [https://en.wikipedia.org/wiki/Algorithms\\_for\\_calculating\\_variance#Welford's\\_Online](https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_Online))

```

updateAggregation :: Measurable → Aggregated → LOMeta → Maybe Int → Aggregated
updateAggregation v (AggregatedStats s) lme resetAfter =
  let count = fcount (fbasic s)
      reset = maybe False (count ≥) resetAfter
  in
  if reset
  then
    singletonStats v
  else
    AggregatedStats $! Stats {flast = v
                              ,fold = mkTimestamp
                              ,fbasic = updateBaseStats (count ≥ 1) v (fbasic s)
                              ,fdelta = updateBaseStats (count ≥ 2) (v - flast s) (fdelta s)
                              ,ftimed = updateBaseStats (count ≥ 2) (mkTimestamp - fold s) (ftimed s)
                              }
  where
    mkTimestamp = utc2ns (tstamp lme)
    utc2ns (UTCTime days secs) =
      let yearsecs :: Rational
          yearsecs = 365 * 24 * 3600
          rdays,rsecs :: Rational
          rdays = toRational $ toModifiedJulianDay days
          rsecs = toRational secs
          s2ns = 1000000000
      in
      Nanoseconds $ round $ (fromRational $ s2ns * rsecs + rdays * yearsecs :: Double)

```



```

updateAggregation v (AggregatedEWMA e) _ _ = AggregatedEWMA $! ewma e v
updateBaseStats :: Bool → Measurable → BaseStats → BaseStats
updateBaseStats False _ s = s {fcount = fcount s + 1}
updateBaseStats True v s =
  let newcount = fcount s + 1
      newvalue = getDouble v
      delta = newvalue - fsum_A s
      dincr = (delta / fromIntegral newcount)
      delta2 = newvalue - fsum_A s - dincr
  in
  BaseStats {fmin = min (fmin s) v
    ,fmax      = max v (fmax s)
    ,fcount    = newcount
    ,fsum_A    = fsum_A s + dincr
    ,fsum_B    = fsum_B s + (delta * delta2)
    }

```

### Calculation of EWMA

Following [https://en.wikipedia.org/wiki/Moving\\_average#Exponential\\_moving\\_average](https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average) we calculate the exponential moving average for a series of values  $Y_t$  according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

The pattern matching below ensures that the **EWMA** will start with the first value passed in, and will not change type, once determined.

```

ewma :: EWMA → Measurable → EWMA
ewma (EmptyEWMA a) v = EWMA a v
ewma (EWMA a s@(Microseconds _)) y@(Microseconds _) =
  EWMA a $ Microseconds $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a s@(Seconds _)) y@(Seconds _) =
  EWMA a $ Seconds $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a s@(Bytes _)) y@(Bytes _) =
  EWMA a $ Bytes $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a (PureI s)) (PureI y) =
  EWMA a $ PureI $ round $ a * (fromInteger y) + (1 - a) * (fromInteger s)
ewma (EWMA a (PureD s)) (PureD y) =
  EWMA a $ PureD $ a * y + (1 - a) * s
ewma _ _ = error "Cannot average on values of different type"

```

### 1.5.31 Cardano.BM.Output.Monitoring

#### Structure of Monitoring

```

type MonitorMVar = MVar MonitorInternal
newtype Monitor = Monitor

```

```

    {getMon :: MonitorMVar}
data MonitorInternal = MonitorInternal
    { monQueue :: TBQ.TBQueue (Maybe NamedLogItem)
    }

```

### Relation from context name to monitoring state

We remember the state of each monitored context name.

```

data MonitorState = MonitorState {
    _expression :: MEvExpr
  , _actions    :: [MEvAction]
  , _environment :: Environment
  }
type MonitorMap = HM.HashMap LoggerName MonitorState

```

### Monitor view is an effectuator

Function *effectuate* is called to pass in a **NamedLogItem** for monitoring.

```

instance IsEffectuator Monitor where
    effectuate monitor item = do
        mon ← readMVar (getMon monitor)
        nocapacity ← atomically $ TBQ.isFullTBQueue (monQueue mon)
        if nocapacity
        then handleOverflow monitor
        else atomically $ TBQ.writeTBQueue (monQueue mon) $ Just item
    handleOverflow _ = putStrLn "Notice: Monitor's queue full, dropping log items!"

```

### Monitor implements Backend functions

Monitor is an **IsBackend**

```

instance IsBackend Monitor where
    typeof _ = MonitoringBK
    realize _ = error "Monitoring cannot be instantiated by 'realize'"
    realizefrom sbtrace@(ctx, _) _ = do
        let config = configuration ctx
        monref ← newEmptyMVar
        let monitor = Monitor monref
        queue ← atomically $ TBQ.newTBQueue 512
        dispatcher ← spawnDispatcher queue config sbtrace
        -- link the given Async to the current thread, such that if the Async
        -- raises an exception, that exception will be re-thrown in the current
        -- thread, wrapped in ExceptionInLinkedThread.
        Async.link dispatcher

```

```

    putMVar monref $ MonitorInternal
      { monQueue = queue
      -- , monState = mempty
      }
    return monitor
  unrealize _ = return ()

```

### Asynchronously reading log items from the queue and their processing

```

spawnDispatcher :: TBQ.TBQueue (Maybe NamedLogItem)
  → Configuration
  → Trace.Trace IO
  → IO (Async.Async ())
spawnDispatcher mqueue config sbtrace = do
  now ← getCurrentTime
  let messageCounters = resetCounters now
  countersMVar ← newMVar messageCounters
  _timer ← Async.async $ sendAndResetAfter
    sbtrace
    "#messagecounters.monitoring"
    countersMVar
    60000 -- 60000 ms = 1 min
    Warning -- Debug
  Async.async (initMap >>= qProc countersMVar)
where
  qProc counters state = do
    maybeItem ← atomically $ TBQ.readTBQueue mqueue
    case maybeItem of
      Just (LogNamed logname logvalue@(LogObject _ _)) → do
        state' ← evalMonitoringAction state logname logvalue
        -- increase the counter for the type of message
        modifyMVar counters $ \cnt → return $ updateMessageCounters cnt logvalue
        qProc counters state'
      Nothing → return () -- stop here
  initMap = do
    ls ← getMonitors config
    return $ HM.fromList $ map (λ(n,(e,as)) → (n,MonitorState e as HM.empty)) $ HM.toList ls

```

### Evaluation of monitoring action

Inspect the log message and match it against configured thresholds. If positive, then run the action on the current state and return the updated state.

```

evalMonitoringAction :: MonitorMap → LoggerName → LogObject → IO MonitorMap
evalMonitoringAction mmap logname logvalue =
  case HM.lookup logname mmap of

```

```

Nothing → return mmap
Just mon@(MonitorState expr acts env0) → do
  let env' = updateEnv env0 logvalue
  if evaluate env' expr
  then do
    now ← getMonotonicTimeNSec
    let env'' = HM.insert "lastalert" (Nanoseconds now) env'
    TIO.putStrLn $ "alert! " <> logname <> " " <> (pack $ show acts) <> " " <> (pack $ show env'')
    return $ HM.insert logname mon { _environment = env'' } mmap
  else return mmap
where
  utc2ns (UTCTime days secs) =
    let yearsecs :: Rational
        yearsecs = 365 * 24 * 3600
        rdays, rsecs :: Rational
        rdays = toRational $ toModifiedJulianDay days
        rsecs = toRational secs
        s2ns = 1000000000
    in
      Nanoseconds $ round $ (fromRational $ s2ns * rsecs + rdays * yearsecs :: Double)
  updateEnv env (LogObject _ (ObserveOpen _)) = env
  updateEnv env (LogObject _ (ObserveDiff _)) = env
  updateEnv env (LogObject _ (ObserveClose _)) = env
  updateEnv env (LogObject lometa (LogValue vn val)) =
    let addenv = HM.fromList [(vn, val)
      , ("timestamp", utc2ns (tstamp lometa))
    ]
    in
      HM.union addenv env
  updateEnv env (LogObject lometa (LogMessage _logitem)) =
    let addenv = HM.fromList [ ("severity", (Severity (severity lometa)))
      -- , ("selection", (liSelection logitem))
      -- , ("message", (liPayload logitem))
      , ("timestamp", utc2ns (tstamp lometa))
    ]
    in
      HM.union addenv env
  updateEnv env (LogObject lometa (AggregatedMessage vals)) =
    let addenv = ("timestamp", utc2ns (tstamp lometa)) : aggs2measurables vals []
    in
      HM.union (HM.fromList addenv) env
  where
    aggs2measurables [] acc = acc
    aggs2measurables ((n, AggregatedEWMA ewma) : r) acc = aggs2measurables r $ (n <> ".avg", avg ewma)
    aggs2measurables ((n, AggregatedStats s) : r) acc = aggs2measurables r $
      (n <> ".mean", PureD ∘ meanOfStats $ fbasic s)
      : (n <> ".flast", flast s)
      : (n <> ".fcount", PureI ∘ fromIntegral ∘ fcount $ fbasic s)

```

```
      :acc  
-- catch all  
updateEnv env _ = env
```

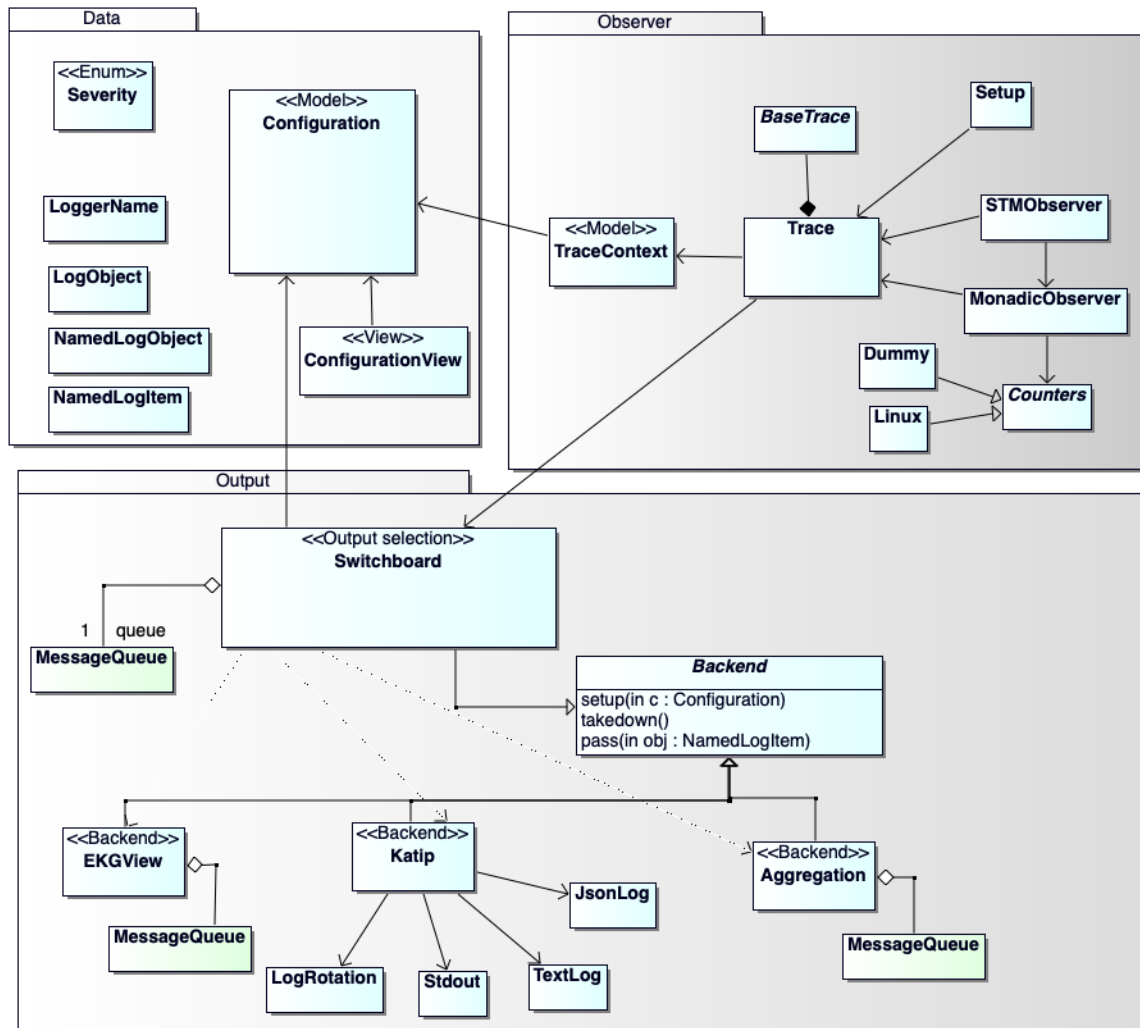


Figure 1.1: Overview of module relationships. The arrows indicate import of a module. The arrows with a triangle at one end would signify "inheritance" in object-oriented programming, but we use it to show that one module replaces the other in the namespace, thus specializes its interface.

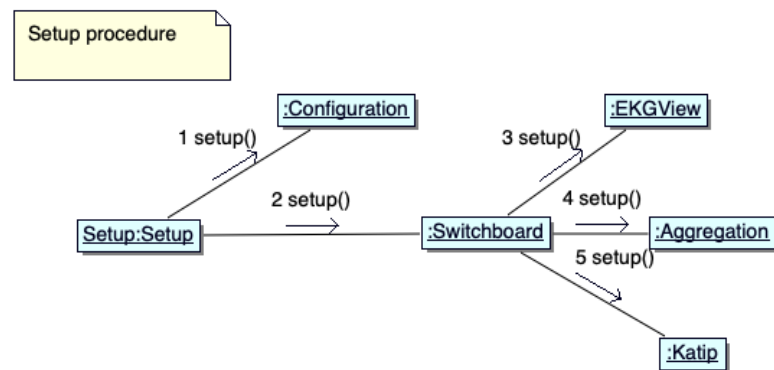


Figure 1.2: Setup procedure

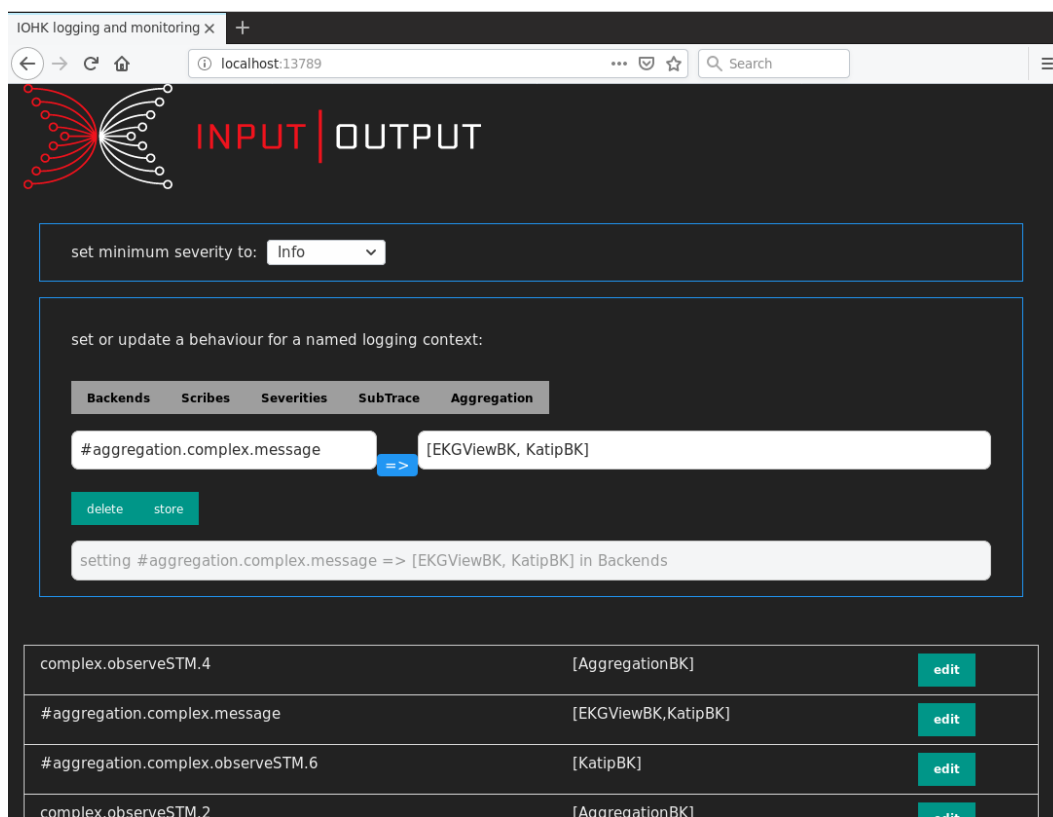


Figure 1.3: The configuration editor is listening on `localhost` and can be accessed through a browser. At the top is the setting for the global minimum severity filter, that drops all messages that have a severity lower than this setting. Below are the settings for various behaviours of the logging system.

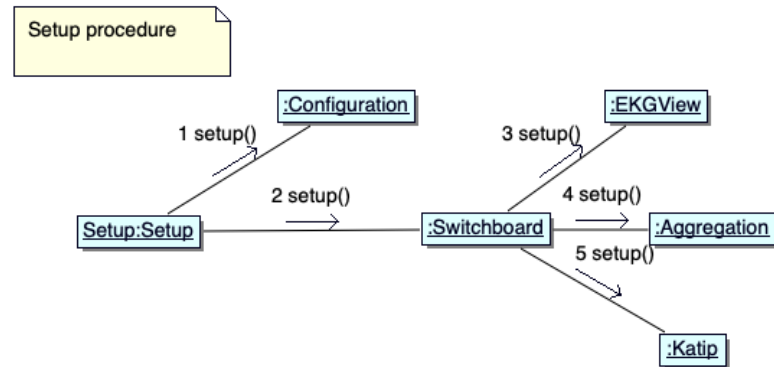


Figure 1.4: Setup procedure

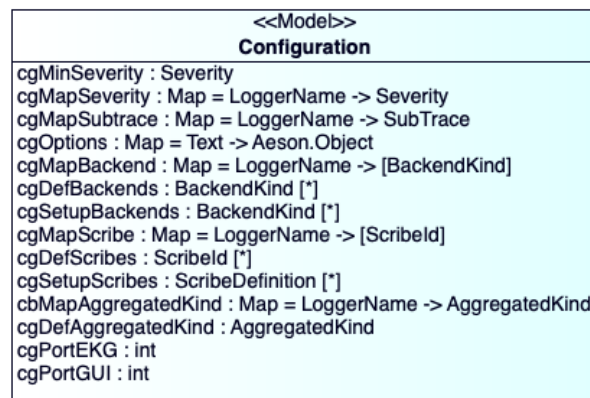


Figure 1.5: Configuration model



# Chapter 2

## Testing

### 2.1 Test 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
```

### 2.2 Test main entry point

```
{-# LANGUAGE CPP #-}
module Main
(
    main
) where
import Test.Tasty
# ifdef ENABLE_AGGREGATION
import qualified Cardano.BM.Test.Agregated (tests)
# endif
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.Rotator (tests)
import qualified Cardano.BM.Test.Routing (tests)
# ifdef ENABLE_MONITORING
import qualified Cardano.BM.Test.Monitoring (tests)
# endif
main :: IO ()
main = defaultMain tests
```

Cardano.BM.Configuration	100%
Cardano.BM.Setup	100%
Cardano.BM.Data.Trace	100%
Cardano.BM.Counters.Common	100%
Cardano.BM.Counters	100%
Cardano.BM.Configuration.Static	100%
Cardano.BM.Configuration.Model	94%
Cardano.BM.Data.MessageCounter	85%
Cardano.BM.Data.Configuration	83%
Cardano.BM.Counters.Linux	81%
Cardano.BM.Output.Switchboard	81%
Cardano.BM.Data.MonitoringEval	81%
Cardano.BM.BaseTrace	80%
Cardano.BM.Observer.Monadic	75%
Cardano.BM.Trace	69%
Cardano.BM.Output.Log	68%
Cardano.BM.Output.Aggregation	68%
Cardano.BM.Data.Aggregated	63%
Cardano.BM.Data.Counter	56%
Cardano.BM.Data.LogItem	53%
Cardano.BM.Data.Backend	50%
Cardano.BM.Rotator	50%
Cardano.BM.Data.BackendKind	50%
Cardano.BM.Data.Output	48%
Cardano.BM.Data.Severity	47%
Cardano.BM.Data.Observable	40%
Cardano.BM.Observer.STM	33%
Cardano.BM.Data.AggregatedKind	33%
Cardano.BM.Data.Rotation	20%
Cardano.BM.Data.SubTrace	10%
Cardano.BM.Output.Monitoring	0%
	63%

Figure 2.1: Test coverage of modules in percent as computed by the tool 'hpc'

```

tests :: TestTree
tests =
  testGroup "iohk-monitoring"
  [
    # ifdef ENABLE_AGGREGATION
      Cardano.BM.Test ◦ Aggregated.tests
    ,
    # endif
    Cardano.BM.Test ◦ STM.tests
    , Cardano.BM.Test ◦ Trace.tests
    , Cardano.BM.Test ◦ Configuration.tests
    , Cardano.BM.Test ◦ Rotator.tests
    , Cardano.BM.Test ◦ Routing.tests
    # ifdef ENABLE_MONITORING
    , Cardano.BM.Test ◦ Monitoring.tests
    # endif
  ]

```

## 2.3 Test case generation

### 2.3.1 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]
    let vs = 42 : 17 : vs'
    ds = map (\(a,b) → a - b) $ zip vs (tail vs)
    (m1,s1) = updateMeanVar $ map fromInteger vs
    (m2,s2) = updateMeanVar $ map fromInteger ds
    mkBasicStats = BaseStats
      (PureI (minimum vs))
      (PureI (maximum vs))
      (fromIntegral $ length vs)
      (m1)
      (s1)
    mkDeltaStats = BaseStats
      (PureI (minimum ds))
      (PureI (maximum ds))
      (fromIntegral $ length ds)
      (m2)
      (s2)
    mkTimedStats = BaseStats
      (Nanoseconds 0)

```

```

      (Nanoseconds 0)
    (0)
    (0)
    (0)
  return $ AggregatedStats (Stats
    (PureI (last vs))
    (Nanoseconds 0)
    mkBasicStats
    mkDeltaStats
    mkTimedStats)

```

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

```

## 2.4 Tests

### 2.4.1 Testing aggregation

```

tests :: TestTree
tests = testGroup "Aggregation measurements" [
  propertyTests
  , unitTests1
  , unitTests2
]

propertyTests :: TestTree
propertyTests = testGroup "Properties" [
  testProperty "minimal" prop_Aggregation_minimal
  , testProperty "commutative" prop_Aggregation_comm
]

unitTests1 :: TestTree
unitTests1 = testGroup "Unit tests for Aggregated" [
  testCase "compare equal >" unitAggregatedEqualGT
  , testCase "compare equal <" unitAggregatedEqualLT
  , testCase "compare different >" unitAggregatedDiffGT
]

```

```

    ,testCase "compare different <" unitAggregatedDiffLT
  ]
unitTests2 :: TestTree
unitTests2 = testGroup "Unit tests for Aggregation" [
  testCase "initial -1" unitAggregationInitialMinus1
  ,testCase "initial +1" unitAggregationInitialPlus1
  ,testCase "initial +0" unitAggregationInitialZero
  ,testCase "initial +1, -1" unitAggregationInitialPlus1Minus1
  ,testCase "stepwise" unitAggregationStepwise
]

```

## Property tests

```

prop_Aggregation_minimal :: Bool
prop_Aggregation_minimal = True

lometa :: LOMeta
lometa = unsafePerformIO $ mkLOMeta Debug

prop_Aggregation_comm :: Integer → Integer → Aggregated → Property
prop_Aggregation_comm v1 v2 ag =
  let AggregatedStats stats1 = updateAggregation (PureI v1) (updateAggregation (PureI v2) ag lometa Nothing)
      AggregatedStats stats2 = updateAggregation (PureI v2) (updateAggregation (PureI v1) ag lometa Nothing)
  in
    fbasic stats1 == fbasic stats2 &&
    (v1 == v2) 'implies' (fstats stats1 == fstats stats2)
-- implication: if p1 is true, then return p2; otherwise true
implies :: Bool → Property → Property
implies p1 p2 = property (¬ p1) .|. p2

```

## Unit tests for Aggregation

```

unitAggregationInitialMinus1 :: Assertion
unitAggregationInitialMinus1 = do
  let AggregatedStats stats1 = updateAggregation (-1) firstStateAggregatedStats lometa Nothing
  flast stats1 @? = (-1)
  (fbasic stats1) @? = BaseStats (-1) 0 2 (-0.5) 0.5
  (fdelta stats1) @? = BaseStats 0 0 1 0 0
  -- AggregatedStats (Stats (-1) 0 (BaseStats (-1) 0 2 (-0.5) 0.5) (BaseStats 0 0 1 0 0))
unitAggregationInitialPlus1 :: Assertion
unitAggregationInitialPlus1 = do
  let AggregatedStats stats1 = updateAggregation 1 firstStateAggregatedStats lometa Nothing
  flast stats1 @? = 1
  (fbasic stats1) @? = BaseStats 0 1 2 0.5 0.5
  (fdelta stats1) @? = BaseStats 0 0 1 0 0
  -- AggregatedStats (Stats 1 0 (BaseStats 0 1 2 0.5 0.5) (BaseStats 0 0 1 0 0))
unitAggregationInitialZero :: Assertion

```

```

unitAggregationInitialZero = do
  let AggregatedStats stats1 = updateAggregation 0 firstStateAggregatedStats lometa Nothing
  flast stats1 @? = 0
  (fbasic stats1) @? = BaseStats 0 0 2 0 0
  (fdelta stats1) @? = BaseStats 0 0 1 0 0
  -- AggregatedStats (Stats 0 0 (BaseStats 0 0 2 0 0) (BaseStats 0 0 1 0 0) (BaseStat
unitAggregationInitialPlus1Minus1 :: Assertion
unitAggregationInitialPlus1Minus1 = do
  let AggregatedStats stats1 = updateAggregation (PureI (-1)) (updateAggregation (PureI 1) firstStateAg
  (fbasic stats1) @? = BaseStats (PureI (-1)) (PureI 1) 3 0.0 2.0
  (fdelta stats1) @? = BaseStats (PureI (-2)) (PureI 0) 2 (-1.0) 2.0
unitAggregationStepwise :: Assertion
unitAggregationStepwise = do
  stats0 ← pure $ singletonStats (Bytes 3000)
  -- putStrLn (show stats0)
  threadDelay 50000 -- 0.05 s
  t1 ← mkLOMeta Debug
  stats1 ← pure $ updateAggregation (Bytes 5000) stats0 t1 Nothing
  -- putStrLn (show stats1)
  -- showTimedMean stats1
  threadDelay 50000 -- 0.05 s
  t2 ← mkLOMeta Debug
  stats2 ← pure $ updateAggregation (Bytes 1000) stats1 t2 Nothing
  -- putStrLn (show stats2)
  -- showTimedMean stats2
  checkTimedMean stats2
  threadDelay 50000 -- 0.05 s
  t3 ← mkLOMeta Debug
  stats3 ← pure $ updateAggregation (Bytes 3000) stats2 t3 Nothing
  -- putStrLn (show stats3)
  -- showTimedMean stats3
  checkTimedMean stats3
  threadDelay 50000 -- 0.05 s
  t4 ← mkLOMeta Debug
  stats4 ← pure $ updateAggregation (Bytes 1000) stats3 t4 Nothing
  -- putStrLn (show stats4)
  -- showTimedMean stats4
  checkTimedMean stats4
where
  checkTimedMean (AggregatedEWMA _) = return ()
  checkTimedMean (AggregatedStats s) = do
    let mean = meanOfStats (ftimed s)
    assertBool "the mean should be >= the minimum" (mean ≥ getDouble (fmin (ftimed s)))
    assertBool "the mean should be <= the maximum" (mean ≤ getDouble (fmax (ftimed s)))

```

commented out:

```

showTimedMean (AggregatedEWMA _) = return ()
showTimedMean (AggregatedStats s) = putStrLn $ "mean = " ++ show (meanOfStats (ftimed s)) ++ showUnits (fm

```

```

firstStateAggregatedStats :: Aggregated
firstStateAggregatedStats = AggregatedStats (Stats z z (BaseStats z z 1 0 0) (BaseStats z z 0 0 0) (BaseStats z z 0 0 0))
  where
    z = PureI 0

```

### Unit tests for Aggregated

```

unitAggregatedEqualGT :: Assertion
unitAggregatedEqualGT = do
  assertBool "comparing seconds"
    ((Seconds 3) > (Seconds 2))
  assertBool "comparing microseconds"
    ((Microseconds 3000) > (Microseconds 2000))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 3000000) > (Nanoseconds 2000000))
  assertBool "comparing bytes"
    ((Bytes 2048) > (Bytes 1024))
  assertBool "comparing doubles"
    ((PureD 2.34) > (PureD 1.42))
  assertBool "comparing integers"
    ((PureI 2) > (PureI 1))
  assertBool "comparing severities"
    ((Severity Error) > (Severity Warning))
unitAggregatedEqualLT :: Assertion
unitAggregatedEqualLT = do
  assertBool "comparing seconds"
    ((Seconds 2) < (Seconds 3))
  assertBool "comparing microseconds"
    ((Microseconds 2000) < (Microseconds 3000))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 2000000) < (Nanoseconds 3000000))
  assertBool "comparing bytes"
    ((Bytes 1024) < (Bytes 2048))
  assertBool "comparing doubles"
    ((PureD 1.34) < (PureD 2.42))
  assertBool "comparing integers"
    ((PureI 1) < (PureI 2))
  assertBool "comparing severities"
    ((Severity Info) < (Severity Notice))
unitAggregatedDiffGT :: Assertion
unitAggregatedDiffGT = do
  assertBool "comparing time (µs vs. s)"
    ((Microseconds 3000000) > (Seconds 2))
  assertBool "comparing time (µs vs. ns)"
    ((Microseconds 30) > (Nanoseconds 29999))
  assertBool "comparing nanoseconds"

```

```

    ((Nanoseconds 3000000) > (Microseconds 2900))
assertBool "comparing bytes"
    ((Bytes 2048) > (PureI 1024))
assertBool "comparing doubles"
    ((PureD 2.34) > (PureI 1))
assertBool "comparing integers"
    ((PureI 2) > (PureD 1.42))
unitAggregatedDiffLT :: Assertion
unitAggregatedDiffLT = do
    assertBool "comparing time (µs vs. s)"
        ((Microseconds 2999999) < (Seconds 3))
    assertBool "comparing time (µs vs. ns)"
        ((Microseconds 30) < (Nanoseconds 30001))
    assertBool "comparing nanoseconds"
        ((Nanoseconds 3000000) < (Microseconds 3001))
    assertBool "comparing bytes"
        ((PureI 1024) < (Bytes 2048))
    assertBool "comparing doubles"
        ((PureD 2.34) < (PureI 3))
    assertBool "comparing integers"
        ((PureI 2) < (PureD 3.42))

```

### 2.4.2 Cardano.BM.Test.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

```

### 2.4.3 Cardano.BM.Test.Trace

```

tests :: TestTree
tests = testGroup "Testing Trace" [
    unit_tests
    , testCase "forked traces stress testing" stressTraceInFork
    , testCase "stress testing: ObservableTrace vs. NoTrace" timingObservableVsUntimed
    , testCaseInfo "demonstrating logging" simpleDemo
]

```



```

,testCaseInfo "demonstrating nested named context logging" exampleWithNamedContexts
]
unit_tests :: TestTree
unit_tests = testGroup "Unit tests" [
  testCase "opening messages should not be traced" unitNoOpeningTrace
, testCase "hierarchy of traces" unitHierarchy
, testCase "forked traces" unitTraceInFork
, testCase "hierarchy of traces with NoTrace" $
  unitHierarchy' [Neutral,NoTrace,(ObservableTrace observablesSet)]
  onlyLevelOneMessage
, testCase "hierarchy of traces with DropOpening" $
  unitHierarchy' [Neutral,DropOpening,(ObservableTrace observablesSet)]
  notObserveOpen
, testCase "hierarchy of traces with UntimedTrace" $
  unitHierarchy' [Neutral,UntimedTrace,UntimedTrace]
  observeNoMeasures
, testCase "changing the minimum severity of a trace at runtime"
  unitTraceMinSeverity
, testCase "changing the minimum severity of a named context at runtime"
  unitNamedMinSeverity
, testCase "appending names should not exceed 80 chars" unitAppendName
, testCase "creat subtrace which duplicates messages" unitTraceDuplicate
, testCase "testing name filtering" unitNameFiltering
, testCase "testing throwing of exceptions" unitExceptionThrowing
, testCase "NoTrace: check lazy evaluation" unitTestLazyEvaluation
, testCase "private messages should not be logged into private files" unitLoggingPrivate
]
where
  observablesSet = [MonotonicClock,MemoryStats]
  notObserveOpen :: [LogObject] → Bool
  notObserveOpen = all (λcase {LogObject _ (ObserveOpen _) → False; _ → True})
  notObserveClose :: [LogObject] → Bool
  notObserveClose = all (λcase {LogObject _ (ObserveClose _) → False; _ → True})
  notObserveDiff :: [LogObject] → Bool
  notObserveDiff = all (λcase {LogObject _ (ObserveDiff _) → False; _ → True})
  onlyLevelOneMessage :: [LogObject] → Bool
  onlyLevelOneMessage = λcase
    [LogObject _ (LogMessage (LogItem _ "Message from level 1."))] → True
    _ → False
  observeNoMeasures :: [LogObject] → Bool
  observeNoMeasures obs = notObserveOpen obs ∧ notObserveClose obs ∧ notObserveDiff obs

```

## 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
  ctx ← liftIO $ newContext name c sev
  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

```

### Simple demo of logging.

```

simpleDemo :: IO String
simpleDemo = do
  cfg ← defaultConfigTesting
  logTrace ← Setup.setupTrace (Right cfg) "test"
  putStrLn "\n"

  logDebug    logTrace "This is how a Debug message likes."
  logInfo     logTrace "This is how an Info message likes."
  logNotice   logTrace "This is how a Notice message likes."
  logWarning  logTrace "This is how a Warning message likes."
  logError    logTrace "This is how an Error message likes."
  logCritical logTrace "This is how a Critical message likes."
  logAlert    logTrace "This is how an Alert message likes."
  logEmergency logTrace "This is how an Emergency message likes."

  return ""

```

### Example of using named contexts with **Trace**

```

exampleWithNamedContexts :: IO String
exampleWithNamedContexts = do
  cfg ← defaultConfigTesting
  Setup.withTrace cfg "test" $ \logTrace → do
    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 100000
    -- force garbage collection to allow exceptions to be thrown
    performMajorGC
    threadDelay 100000

    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 Debug "STM-action" setVar_
        logInfo trInner "let's see: done."

```

### Show effect of turning off observables

```

runTimedAction :: Trace IO → Int → IO Measurable
runTimedAction logTrace reps = do
    runid ← newUnique
    t0 ← getMonoClock
    forM_ [(1 :: Int)..reps] $ const $ observeAction logTrace
    t1 ← getMonoClock
    return $ diffTimeObserved (CounterState runid t0) (CounterState runid t1)
where
    observeAction trace = do
        _ ← MonadicObserver.bracketObserveIO trace Debug "" action
        return ()
    action = return $ forM [1 :: Int..100] $ \x → [x] ++ (init $ reverse [1 :: Int..10000])

timingObservableVsUntimed :: Assertion
timingObservableVsUntimed = 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 ← runTimedAction traceObservable 100
    t_untimed ← runTimedAction traceUntimed 100
    t_notrace ← runTimedAction traceNoTrace 100
    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, GhcRtsStats, MemoryStats, IOStats, ProcessStats]

```

### 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.

```

unitHierarchy :: Assertion
unitHierarchy = 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.

```

unitTraceMinSeverity :: Assertion
unitTraceMinSeverity = 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
      (λcase
        LogObject (LOMeta _ _ Info) (LogMessage (LogItem _ "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.

```

unitTraceDuplicate::Assertion
unitTraceDuplicate = 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.

```

unitNamedMinSeverity::Assertion
unitNamedMinSeverity = 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
    LogObject (LOMeta _ _ Info) (LogMessage (LogItem _ "Message #2")) → False
    _ → True)
  res)

unitHierarchy' :: [SubTrace] → ([LogObject] → Bool) → Assertion
unitHierarchy' 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 Debug "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

```

unitTraceInFork :: Assertion
unitTraceInFork = 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

```

stressTraceInFork :: Assertion
stressTraceInFork = 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

```

unitNoOpeningTrace :: Assertion
unitNoOpeningTrace = do
  msgs ← STM.newTVarIO [ ]
  logTrace ← setupTrace $ TraceConfiguration (TVarList msgs) "test" DropOpening Debug
  _ ← STMObserver.bracketObserveIO logTrace Debug "setTVar" setVar_

```



```

res ← STM.readTVarIO msgs
assertBool
  ("Found non-expected ObserveOpen message: " ++ show res)
  (all (λcase (LogObject _ (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 80.

```

unitAppendName :: Assertion
unitAppendName = do
  cfg ← defaultConfigTesting
  Setup.withTrace cfg "test" $ λtrace0 → do
    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 "abcdefghijklmnopqrstuvwxy"

```

```

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

```

unitNameFiltering :: Assertion
unitNameFiltering = do
  let contextName = "test.sub.1"
  let lname = "sum" -- would be part of a "LogValue lname 42"
  let filter1 = [(Drop (Exact "test.sub.1"), Unhide [ ])]
  assertBool ("Dropping a specific name should filter it out and thus return False")
    (False ≡ evalFilters filter1 contextName)
  let filter2 = [(Drop (EndsWith ".1"), Unhide [ ])]
  assertBool ("Dropping a name ending with a specific text should filter out the context name")
    (False ≡ evalFilters filter2 contextName)
  let filter3 = [(Drop (StartsWith "test."), Unhide [ ])]
  assertBool ("Dropping a name starting with a specific text should filter out the context name")
    (False ≡ evalFilters filter3 contextName)
  let filter4 = [(Drop (Contains ".sub."), Unhide [ ])]
  assertBool ("Dropping a name starting containing a specific text should filter out the context name")
    (False ≡ evalFilters filter4 contextName)

```

```

    (False == evalFilters filter4 contextName)
  let filter5 = [(Drop (StartsWith "test."),
    Unhide [(Exact "test.sub.1")])]
  assertBool ("Dropping all and un hiding a specific name should the context name allow passing")
    (True == evalFilters filter5 contextName)
  let filter6 = [(Drop (StartsWith "test."),
    Unhide [(EndsWith ".sum"),
      (EndsWith ".other")])]
  assertBool ("Dropping all and un hiding some names, the LogObject should pass the filter")
    (True == evalFilters filter6 (contextName <> "." <> lname))
  let filter7 = [(Drop (StartsWith "test."),
    Unhide [(EndsWith ".product")])]
  assertBool ("Dropping all and un hiding an inexistant named value, the LogObject should not pass")
    (False == evalFilters filter7 (contextName <> "." <> lname))
  let filter8 = [(Drop (StartsWith "test."),
    Unhide [(Exact "test.sub.1")],
    (Drop (StartsWith "something.else."),
      Unhide [(EndsWith ".this")])])
  assertBool ("Disjunction of filters that should pass")
    (True == evalFilters filter8 contextName)
  let filter9 = [(Drop (StartsWith "test."),
    Unhide [(Exact ".that")],
    (Drop (StartsWith "something.else."),
      Unhide [(EndsWith ".this")])])
  assertBool ("Disjunction of filters that should not pass")
    (False == evalFilters filter9 contextName)

```

## Exception throwing

Exceptions encountered should be thrown.

```

unitExceptionThrowing :: Assertion
unitExceptionThrowing = do
  action ← work msg
  res ← Async.waitCatch action
  assertBool
    ("Exception should have been rethrown")
    (isLeft res)
where
  msg :: Text
  msg = error "faulty message"
  work :: Text → IO (Async.Async ())
  work message = Async.async $ do
    cfg ← defaultConfigTesting
    trace ← Setup.setupTrace (Right cfg) "test"
    logInfo trace message
    threadDelay 10000

```

**Check lazy evaluation of trace**

Exception should not be thrown when type of `Trace` is `NoTrace`.

```

unitTestLazyEvaluation :: Assertion
unitTestLazyEvaluation = do
    action ← work msg
    res ← Async.waitCatch action
    assertBool
        ("Exception should not have been rethrown when type of Trace is NoTrace")
        (isRight res)
where
    msg :: Text
    msg = error "faulty message"
    work :: Text → IO (Async.Async ())
    work message = Async.async $ do
        cfg ← defaultConfigTesting
        trace0@(ctx, _) ← Setup.setupTrace (Right cfg) "test"
        setSubTrace (configuration ctx) "test.work" (Just NoTrace)
        trace ← subTrace "work" trace0
        logInfo trace message

```

**Check that private messages do not end up in public log files.**

```

unitLoggingPrivate :: Assertion
unitLoggingPrivate = do
    tmpDir ← getTemporaryDirectory
    let privateFile = tmpDir </> "private.log"
        publicFile = tmpDir </> "public.log"
    conf ← empty
    setDefaultBackends conf [KatipBK]
    setSetupBackends conf [KatipBK]
    setDefaultScribes conf [ "FileTextSK:" <> pack privateFile
                            , "FileTextSK:" <> pack publicFile
                            ]
    setSetupScribes conf [ ScribeDefinition
        { scKind    = FileTextSK
        , scName    = pack privateFile
        , scPrivacy = ScPrivate
        , scRotation = Nothing
        }
    , ScribeDefinition
        { scKind    = FileTextSK
        , scName    = pack publicFile
        , scPrivacy = ScPublic
        , scRotation = Nothing
    }

```

```

    }
  ]
Setup.withTrace conf "test" $ \trace → do
  -- should log in both files
  logInfo trace message
  -- should only log in private file
  logInfoS trace message
countPublic ← length ∘ lines < $ > readFile publicFile
countPrivate ← length ∘ lines < $ > readFile privateFile
-- delete files
forM_ [privateFile, publicFile] removeFile
assertBool
  ("Private file should contain 2 lines and it contains " ++ show countPrivate ++ ".\n" ++
   "Public file should contain 1 line and it contains " ++ show countPublic ++ ".\n"
  )
  (countPublic ≡ 1 ∧ countPrivate ≡ 2)
where
  message :: Text
  message = "Just a message"

```

#### 2.4.4 Testing configuration

##### Test declarations

```

tests :: TestTree
tests = testGroup "config tests" [
  propertyTests
, unitTests
]
propertyTests :: TestTree
propertyTests = testGroup "Properties" [
  testProperty "minimal" prop_Configuration_minimal
]
unitTests :: TestTree
unitTests = testGroup "Unit tests" [
  testCase "static representation" unitConfigurationStaticRepresentation
, testCase "parsed representation" unitConfigurationParsedRepresentation
, testCase "parsed configuration" unitConfigurationParsed
, testCase "include EKG if defined" unitConfigurationCheckEKGpositive
, testCase "not include EKG if not def" unitConfigurationCheckEKGnegative
, testCase "check scribe caching" unitConfigurationCheckScribeCache
, testCase "test ops on Configuration" unitConfigurationOps
]

```

### 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.

```
unitConfigurationCheckEKGpositive :: Assertion
unitConfigurationCheckEKGpositive = do
  # ifdef ENABLE_EKG
    return ()
  # else
    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)
  # endif
```

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

```
unitConfigurationCheckEKGnegative :: Assertion
unitConfigurationCheckEKGnegative = do
```

```

# ifdef ENABLE_EKG
  return ()
# else
  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)
# endif

```

```

unitConfigurationStaticRepresentation :: Assertion
unitConfigurationStaticRepresentation =
  let r = Representation
    { minSeverity = Info
    , rotation = Just $ RotationParameters
      { rpLogLimitBytes = 5000000
      , rpMaxAgeHours = 24
      , rpKeepFilesNum = 10
      }
    , setupScribes =
      [ ScribeDefinition { scName = "stdout"
      , scKind = StdoutSK

```

```

        ,scPrivacy = ScPublic
        ,scRotation = Nothing}
    ]
  ,defaultScribes = [(StdoutSK, "stdout")]
  ,setupBackends = [EKGVViewBK, 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 @? =
  (intercalate "\n"
    [ "rotation:"
    , "  rpLogLimitBytes: 5000000"
    , "  rpKeepFilesNum: 10"
    , "  rpMaxAgeHours: 24"
    , "defaultBackends:"
    , "- KatipBK"
    , "setupBackends:"
    , "- EKVViewBK"
    , "- KatipBK"
    , "hasGUI: 12789"
    , "defaultScribes:"
    , "- - StdoutSK"
    , "  - stdout"
    , "options:"
    , "  test2:"
    , "    value: object2"
    , "  test1:"
    , "    value: object1"
    , "setupScribes:"
    , "- scName: stdout"
    , "  scRotation: null"
    , "  scKind: StdoutSK"
    , "  scPrivacy: ScPublic"
    , "hasEKG: 18321"
    , "minSeverity: Info"
    , "-- to force a line feed at the end of the file"
    ]
  )
unitConfigurationParsedRepresentation :: Assertion
unitConfigurationParsedRepresentation = do
  repr ← parseRepresentation "test/config.yaml"
  encode repr @? =

```

```

(intercalate "\n"
["rotation:"
, "  rpLogLimitBytes: 5000000"
, "  rpKeepFilesNum: 10"
, "  rpMaxAgeHours: 24"
, "defaultBackends:"
, "- KatipBK"
, "setupBackends:"
, "- AggregationBK"
, "- EKGViewBK"
, "- KatipBK"
, "hasGUI: null"
, "defaultScribes:"
, "- - StdoutSK"
, "  - stdout"
, "options:"
, "  mapSubtrace:"
, "    iohk.benchmarking:"
, "      tag: ObservableTrace"
, "      contents:"
, "        - GhcRtsStats"
, "        - MonotonicClock"
, "    iohk.deadend: NoTrace"
, "  mapSeverity:"
, "    iohk.startup: Debug"
, "    iohk.background.process: Error"
, "    iohk.testing.uncritical: Warning"
, "  mapAggregatedkinds:"
, "    iohk.interesting.value: EwmaAK {alpha = 0.75}"
, "    iohk.background.process: StatsAK"
, "  cfokey:"
, "    value: Release-1.0.0"
, "  mapMonitors:"
, "    chain.creation.block:"
, "      - monitor: ((time > (23 s)) Or (time < (17 s)))"
, "      - actions:"
, "        - AlterMinSeverity \"chain.creation\" Debug"
, "        ! '#aggregation.critproc.observable':"
, "      - monitor: (mean >= (42))"
, "      - actions:"
, "        - CreateMessage \"exceeded\" \"the observable has been too long too high!\""
, "        - AlterGlobalMinSeverity Info"
, "  mapScribes:"
, "    iohk.interesting.value:"
, "      - StdoutSK::stdout"
, "      - FileTextSK::testlog"
, "    iohk.background.process: FileTextSK::testlog"
, "  mapBackends:"

```



```

    , "    iohk.interesting.value:"
    , "    - EKGViewBK"
    , "    - AggregationBK"
    , "setupScribes:"
    , "- scName: testlog"
    , "    scRotation:"
    , "        rpLogLimitBytes: 25000000"
    , "        rpKeepFilesNum: 3"
    , "        rpMaxAgeHours: 24"
    , "    scKind: FileTextSK"
    , "    scPrivacy: ScPrivate"
    , "- scName: stdout"
    , "    scRotation: null"
    , "    scKind: StdoutSK"
    , "    scPrivacy: ScPublic"
    , "hasEKG: 12789"
    , "minSeverity: Info"
    , "-- to force a line feed at the end of the file
  ]
)

```

*unitConfigurationParsed* :: Assertion

*unitConfigurationParsed* = **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" )])

    , ("mapMonitors", *HM.fromList* [ ("chain.creation.block", *Array* \$ *V.fromList*

        [ *Object* (*HM.fromList* [ ("monitor", *String* "((time > (23 s)) Or (time < (17 s))

        , *Object* (*HM.fromList* [ ("actions", *Array* \$ *V.fromList*

            [ *String* "AlterMinSeverity \chain.creation\ Debug" ])))

    , ("#aggregation.critproc.observable", *Array* \$ *V.fromList*

```

[Object (HM.fromList [("monitor",String "(mean >= (42))")])
, Object (HM.fromList [("actions",Array$ V.fromList
  [String "CreateMessage \"exceeded\" \"the observable has been too
    ,String "AlterGlobalMinSeverity Info"])]))]
,("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"])
  ]
,cgMapScribeCache  = HM.fromList [("iohk.interesting.value",
  ["StdoutSK::stdout","FileTextSK::testlog"])
  ,("iohk.background.process",["FileTextSK::testlog"])
  ]
,cgDefScribes      = ["StdoutSK::stdout"]
,cgSetupScribes    = [ScribeDefinition
  {scKind      = FileTextSK
  ,scName      = "testlog"
  ,scPrivacy   = ScPrivate
  ,scRotation  = Just$RotationParameters
    {rpLogLimitBytes = 25000000

```

```

        ,rpMaxAgeHours = 24
        ,rpKeepFilesNum = 3
    }
}
,ScribeDefinition
{scKind = StdoutSK
 ,scName = "stdout"
 ,scPrivacy = ScPublic
 ,scRotation = Nothing
}
]
,cgMapAggregatedKind = HM.fromList [("iohk.interesting.value",EwmaAK {alpha = 0.75})
 ,("iohk.background.process",StatsAK)
]
,cgDefAggregatedKind = StatsAK
,cgMonitors = HM.fromList [("chain.creation.block",((OR (Compare "time" ((>),(Agg.Second
 ,["AlterMinSeverity \"chain.creation\" Debug"]
)
)
)
 ,("#aggregation.critproc.observable",(Compare "mean" ((≥),(Agg.PureI 42))
 ,["CreateMessage \"exceeded\" \"the observable has been too long t
 ,["AlterGlobalMinSeverity Info"]
)
)
)
]
,cgPortEKG = 12789
,cgPortGUI = 0
}

```

Test caching and inheritance of Scribes.

```

unitConfigurationCheckScribeCache :: Assertion
unitConfigurationCheckScribeCache = do
    configuration ← empty
    let defScribes = ["FileTextSK::node.log"]
    setDefaultScribes configuration defScribes
    let scribes12 = ["StdoutSK::stdout", "FileTextSK::out.txt"]
    setScribes configuration "name1.name2" $ Just scribes12
    scribes1234 ← getScribes configuration "name1.name2.name3.name4"
    scribes1 ← getScribes configuration "name1"
    scribes1234cached ← getCacheScribes configuration "name1.name2.name3.name4"
    scribesXcached ← getCacheScribes configuration "nameX"
    assertBool "Scribes for name1.name2.name3.name4 must be the same as name1.name2" $
        scribes1234 == scribes12
    assertBool "Scribes for name1 must be the default ones" $
        scribes1 == defScribes
    assertBool "Scribes for name1.name2.name3.name4 must have been cached" $

```

```

scribes1234cached ≡ Just scribes1234
assertBool "Scribes for nameX must not have been cached since getScribes was not called" $
  scribesXcached ≡ Nothing

```

Test operations on Configuration.

```

unitConfigurationOps :: Assertion
unitConfigurationOps = do
  configuration ← defaultConfigStdout
  defBackends ← getDefaultBackends configuration
  setDefaultAggregatedKind configuration $ EwmaAK 0.01
  -- since loggername does not exist the default must be inherited
  defAggregatedKind ← getAggregatedKind configuration "non-existent loggername"
  setAggregatedKind configuration "name1" $ Just StatsAK
  name1AggregatedKind ← getAggregatedKind configuration "name1"
  setEKGport configuration 11223
  ekgPort ← getEKGport configuration
  setGUIport configuration 1080
  guiPort ← getGUIport configuration
  assertBool "Default backends" $
    defBackends ≡ [KatipBK]
  assertBool "Default aggregated kind" $
    defAggregatedKind ≡ EwmaAK 0.01
  assertBool "Specific name aggregated kind" $
    name1AggregatedKind ≡ StatsAK
  assertBool "Set EKG port" $
    ekgPort ≡ 11223
  assertBool "Set GUI port" $
    guiPort ≡ 1080

```

### 2.4.5 Rotator

```

tests :: TestTree
tests = testGroup "testing Trace" [
  property_tests
]
property_tests :: TestTree
property_tests = testGroup "Property tests" [
  testProperty "rotator: name giving" propNameGiving
  , testProperty "rotator: cleanup" $ propCleanup $ rot n
]
where
  n = 5
  rot num = RotationParameters
    { rpLogLimitBytes = 10000000 -- 10 MB

```

```

    ,rpMaxAgeHours = 24
    ,rpKeepFilesNum = num
  }

```

Check that full file name has only added 15 digits to the base name of the file.

```

propNameGiving :: FilePath → Property
propNameGiving name = ioProperty $ do
  filename ← nameLogFile name
  return $ length filename == length name + 15

```

Test cleanup of rotator.

This test creates a random number of files with the same name but with different dates and afterwards it calls the `cleanupRotator` function which removes old log files keeping only `rpKeepFilesNum` files and deleting the others.

```

data LocalFilePath = Dir FilePath
  deriving (Show)
instance Arbitrary LocalFilePath where
  arbitrary = do
    start ← QC.sized $ λn → replicateM (n + 1) (QC.elements $ [ 'a' .. 'z' ])
    x ← QC.sized $ λn → replicateM n (QC.elements $ [ 'a' .. 'd' ] ++ "/" )
    pure $ Dir $ start ++ removeAdjacentAndLastSlashes x
  shrink (Dir path) = map (Dir ∘ removeAdjacentAndLastSlashes ∘ (intercalate "/")) $
    product' $ map (filter (≠ "/")) $ map QC.shrink (splitOn "/" path)
  where
    product' :: [[a]] → [[a]]
    product' = mapM (λx → x >> return)
  removeAdjacentAndLastSlashes :: FilePath → FilePath
  removeAdjacentAndLastSlashes = concat ∘ filter (≠ "/") ∘ groupBy (\_b → b ≠ '/' )
data SmallAndLargeInt = SL Int
  deriving (Show)
instance Arbitrary SmallAndLargeInt where
  arbitrary = do
    QC.oneof [smallGen
      ,largeGen
    ]
  where
    smallGen :: QC.Gen SmallAndLargeInt
    smallGen = do
      QC.Small x ← (QC.arbitrary :: QC.Gen (QC.Small Int))
      pure $ SL $ abs x
    largeGen :: QC.Gen SmallAndLargeInt
    largeGen = do

```

```

    let maxBoundary = 0010000000000000 -- 10 years for the format which is used
    minBoundary = 0000000000100000 -- 1 hour for the format which is used
    x ← QC.choose (minBoundary, maxBoundary)
    pure $ SL x
shrink _ = []
propCleanup :: RotationParameters → LocalFilePath → Positive Int → SmallAndLargeInt → Property
propCleanup rotationParams (Dir filename) (Positive nFiles) (SL maxDev) = ioProperty $ do
    tmpDir ← getTemporaryDirectory
    let path = tmpDir </> filename
    -- generate nFiles different dates
    now ← getCurrentTime
    let tsnow = formatTime defaultTimeLocale tsformat now
    deviations ← replicateM nFiles $ QC.generate $ QC.choose (1, maxDev + 1)
    -- TODO if generated within the same sec we have a problem
    let dates = map show $ scanl (+) (read tsnow) deviations
        files = map (\a → path ++ ('-' : a)) dates
        sortedFiles = reverse $ sort files
        keepFilesNum = fromIntegral $ rpKeepFilesNum rotationParams
        toBeKept = reverse $ take keepFilesNum sortedFiles
    createDirectoryIfMissing True $ takeDirectory path
    forM_ (files) $ \f → openFile f WriteMode
    cleanupRotator rotationParams path
    filesRemained ← listLogFiles path
    let kept = case filesRemained of
        Nothing → []
        Just l → NE.toList l
    -- delete the files left
    forM_ kept removeFile
    -- delete folders created
    when (dropWhile (≠ '/') filename ≠ "") $
        removePathForcibly $ "/" tmp </> takeWhile (≠ '/') filename
    return $ kept == toBeKept

```

# Index

- Aggregated, 42
  - instance of Semigroup, 42
  - instance of Show, 43
- AggregatedExpanded, 82
- AggregatedKind, 43
  - EwmaAK, 43
  - StatsAK, 43
- AggregatedMessage, 48
- Aggregation, 82
  - instance of IsBackend, 83
  - instance of IsEffectuator, 83
- appendName, 22
- Backend, 44
- BackendKind, 44
  - AggregationBK, 44
  - EKGViewBK, 44
  - KatipBK, 44
  - MonitoringBK, 44
  - SwitchboardBK, 44
- BaseTrace, 21
  - instance of Contravariant, 21
- Counter, 46
- Counters
  - Dummy
    - readCounters, 30
  - Linux
    - readCounters, 30
- CounterState, 46
- CounterType, 46
- diffCounters, 47
- diffTimeObserved, 28
- EKGView, 78
  - instance of IsBackend, 80
  - instance of IsEffectuator, 79
- evalFilters, 23
- EWMA, 42
- ewma, 87
- getMonoClock, 29
- getOptionOrDefault, 52
- IsBackend, 44
- IsEffectuator, 43
- LOContent, 48
- Log, 70
  - instance of IsBackend, 71
  - instance of IsEffectuator, 70
- logAlert, 25
- logAlertS, 25
- logCritical, 25
- logCriticalS, 25
- logDebug, 25
- logDebugS, 25
- logEmergency, 25
- logEmergencyS, 25
- logError, 25
- logErrorS, 25
- LoggerName, 47
- logInfo, 25
- logInfoS, 25
- LogItem, 49
  - liPayload, 49
  - liSelection, 49
- LogMessage, 48
- LogNamed, 48
- logNotice, 25
- logNoticeS, 25
- LogObject, 48
- LogSelection, 49
  - Both, 49
  - Private, 49
- LogValue, 48
- logWarning, 25
- logWarningS, 25
- LOMeta, 48
- mainTrace, 66
- Measurable, 37

- instance of Num, 39
  - instance of Show, 40
- modifyName, 22
- Monitor, 87
  - instance of IsBackend, 88
  - instance of IsEffectuator, 88
- nameCounter, 46
- NamedLogItem, 47
- natTrace, 21
- newContext, 27
- nominalTimeToMicroseconds, 29
- noTrace, 21
- ObservableInstance, 49
  - GhcRtsStats, 49
  - IOStats, 49
  - MemoryStats, 49
  - MonotonicClock, 49
  - NetStats, 49
  - ProcessStats, 49
- ObserveClose, 48
- ObserveDiff, 48
- ObserveOpen, 48
- OutputKind, 49
  - TVarList, 49
  - TVarListNamed, 49
- parseRepresentation, 45
- Port, 44
- readRTSStats, 29
- Representation, 44
- RotationParameters, 50
  - rpKeepFilesNum, 50
  - rpLogLimitBytes, 50
  - rpMaxAgeHours, 50
- ScribeDefinition, 50
  - scKind, 50
  - scName, 50
  - scPrivacy, 50
  - scRotation, 50
- ScribeId, 50
- ScribeKind
  - FileJsonSK, 49
  - FileTextSK, 49
  - StderrSK, 49
  - StdoutSK, 49
- ScribePrivacy, 50
  - ScPrivate, 50
  - ScPublic, 50
- setupTrace, 27
- Severity, 51
  - Alert, 51
  - Critical, 51
  - Debug, 51
  - Emergency, 51
  - Error, 51
  - Info, 51
  - instance of FromJSON, 51
  - Notice, 51
  - Warning, 51
- shutdown, 27
- singletonStats, 42
- Stats, 40
  - instance of Semigroup, 41
- stats2Text, 41
- stdoutTrace, 24
- SubTrace, 51
  - DropOpening, 51
  - FilterTrace, 51
    - NameOperator, 51
    - NameSelector, 51
  - Neutral, 51
  - NoTrace, 51
  - ObservableTrace, 51
  - TeeTrace, 51
  - UntimedTrace, 51
- subTrace, 26
- Switchboard, 66
  - instance of IsBackend, 67
  - instance of IsEffectuator, 66
  - setupBackends, 68
- Trace, 52
- traceConditionally, 25
- TraceContext, 52
  - configuration, 52
  - loggerName, 52
  - minSeverity, 52
  - tracetype, 52
- traceInTVar, 24
- traceInTVarIO, 24
- TraceNamed, 52
- traceNamedInTVarIO, 24
- traceNamedItem, 25



`traceNamedObject`, 23  
`traceWith`, 21  
`typeofTrace`, 22  
  
`updateAggregation`, 86  
`updateTracetype`, 22  
  
`withTrace`, 27