

# UR2SQL.lhs

Jost Berthold

August 31, 2009

To compile this file: `$ghc --make -O2 UR2SQL.lhs` L<sup>A</sup>T<sub>E</sub>X can be used to produce documentation: use the fancyvrb package, define a verbatim environment named “code”, and a command `\cd` (similar to `\texttt`). and add some frame around this file (begin document and such). You can use `\input`.

Read usage records, convert data if necessary, insert into a database.

This file was written as a top-down design prototype, should phps. be split into several modules later. Some functionality, however, needs to be in tight sync: Parsing functions and SQL database schema are connected.

## Importing libraries, skip for “the big picture”

This imports list should be reduced to what is actually needed later (ghc has an option `-ddump-minimal-imports` to help).

We are using the HDBC library for database access from Haskell, and its ODBC driver, as well as the light-weight XML library XML.Light. Both available from [hackage.haskell.org](http://hackage.haskell.org)<sup>1</sup>.

---

*compiled code*

---

```
{-# OPTIONS -cpp #-}
module Main where

-- dependencies to haskell libraries (not included in GHC)
import Database.HDBC      -- interface
import Database.HDBC.ODBC -- driver
import Text.XML.Light     -- tiny XML tool set

-- Haskell libraries included in GHC:
import System
import Data.Time      -- requires extra instance SqlData for UTCTime

import System.Time    -- old stuff, to be replaced by Data.Time

import System.Locale -- locale for Time libraries
import Data.Maybe
import Data.Char
import Control.Monad.State
import qualified Control.Exception as E
import System.IO

-- debugging
#ifdef TRACE
import Debug.Trace -- logs to stderr when evaluating expressions...
#else
trace _ x = x
```

---

<sup>1</sup>The current version of this code uses library versions suitable for GHC-6.8.3, not the latest GHC-6.11 (6.12 when released). APIs should not change too much, but be aware.

```
#endif
```

---

This is what we want to do:

---

```
usage :: IO()
usage = do n <- getProgName
         putStrLn ("Usage: #>" ++ n ++ " <usage record file>")
         putStrLn (" or #>" ++ n ++ "--create-table")

main = do args <- getArgs
         if null args
         then usage
         else case head args of
              "--create-table" -> putStrLn initTableStatement
              ('-':_)          -> usage
              file              -> do
                  urFile <- readFile file
                  let usageRecords = parseXMLDoc urFile
                      decodedURs  = case usageRecords of
                          Nothing -> error ("Parsing " ++ head args ++ " failed. "
                                           ++ "Not a usage record file?")
                          Just urs  -> map decodeUR (splitURs urs)
                  db <- connectODBC databaseODBC
                  insertAllInto db decodedURs
                  disconnect db

#ifdef DEBUG
         where connectODBC dbstring = do putStrLn ("### connect "
                                                  ++ dbstring ++ " ###")
                                                  return (take 20 dbstring)
         disconnect db = putStrLn ("### disconnect " ++ db ++ " ###")
#endif
```

---

## Usage records as a Haskell data structure

This data structure needs to be filled. Here is the point where XML unmarshalling and database schema need to fit together. Thus, we should not modularise this part of the code, or at least keep this section together in one module.

Every optional value is simply a `Maybe` type, translated into `NULL` by the SQL interface later.

---

```
data UsageRecord = UsageRecord { urId      :: String -- mandatory
                                , urCreateTime :: Maybe UTCTime
                                -- mandatory for us, but defaulted
                                , urStatus   :: JobStatus -- mandatory
                                , urUser     :: User -- mandatory for us
                                           -- (GlobalUserName)
                                , urJob      :: Job
                                , urStartTime :: Maybe UTCTime
                                , urEndTime  :: Maybe UTCTime
                                , urWallDuration :: Maybe Double
                                , urCPUDuration :: Maybe Double -- (not MiG)
                                , urCharge   :: Maybe Double
                                , urNodeCount :: Maybe Int
                                , urMachineName :: Maybe String -- cluster
                                , urQueue    :: Maybe String -- (not MiG)
                                , urHost     :: Maybe String -- node name
                                , urXml      :: String -- verbatim XML
                                }
    deriving (Read, Eq, Show)
```

---

```

data User = User { uGlobal ::String      -- global user name mandatory
                  , uLocal ::Maybe String -- optional user on machine
                  , userVO ::Maybe String -- optional VO, as URL string
                  }
                  deriving (Read, Eq, Show)

data Job = Job { jGlobal, jLocal ::Maybe String }
              deriving (Read, Eq, Show)

data JobStatus = Aborted  -- canceled by user or something external
                | Other String -- for recording unexpected cases.
                                -- We don't want to see these, but...
                | Failed   -- executed, produced failure
                | Completed -- vanilla case
                deriving (Eq, Ord)

instance Show JobStatus where
  show Aborted  = "aborted"
  show (Other s) = s
  show Failed   = "failed"
  show Completed = "completed"

instance Read JobStatus where
  readsPrec _ x | s `startsWith` "aborted" = [(Aborted, drop 7 s)]
                | s `startsWith` "completed" = [(Completed, drop 9 s)]
                | s `startsWith` "failed" = [(Failed, drop 6 s)]
                | otherwise = [(Other (takeWhile (not . isSpace) s),
                                     dropWhile (not . isSpace) s)]
    where s = dropWhile isSpace x
          startsWith _ [] = True
          startsWith [] _ = False
          startsWith (x:str) (m:match)
            | x /= m = False
            | otherwise = startsWith str match

instance SqlType JobStatus where
  toSql = toSql . show
  fromSql = read . fromSql

```

---

The Haskell type should map to the database schema (the respective table for Usage Records). It is, however, possible to

- have additional unused (NULL) fields in the database table
- have fields in the usage record structure which are not intended to reach the database

The connecting point is the insert statement (values used) and the `extractFields` function, which is expected to produce a list of matching length for the prepared statement to insert the values.

---

```

-- fields count and types should fit the insert statement, which
-- should in turn fit the DB schema.
-- if we could reify the data types inside UsageRecord, this could be
-- automatic. For now, it has to be modified manually when necessary.
fields :: [(String,String)] -- list of field names and types
fields = [ ("RecordId"      ,"VARCHAR(512) NOT NULL")
          , ("CreateTime"  ,"TIMESTAMP DEFAULT CURRENT_TIMESTAMP")
          , ("GlobalUserName","VARCHAR(512) NOT NULL")
          , ("LocalUserId"  ,"VARCHAR(512)")
          ]

```

```

-- encoding all VO information in one URL string
-- not provided by MiG
, ("VO" , "VARCHAR(512)")
, ("GlobalJobId" , "VARCHAR(512)")
, ("LocalJobId" , "VARCHAR(512)")
, ("Status" , "VARCHAR(32) NOT NULL") -- value restriction!
, ("StartTime" , "DATETIME") -- optional
, ("EndTime" , "DATETIME") -- optional
, ("WallDuration" , "DOUBLE") -- parsed from a duration
-- not provided by MiG
, ("CPUDuration" , "DOUBLE") -- parsed from a duration
, ("Charge" , "DOUBLE")
, ("NodeCount" , "SMALLINT")
, ("MachineName" , "VARCHAR(256)") -- max len for dnsnames
-- not provided by MiG
, ("Queue" , "VARCHAR(64)") -- simple short name
, ("Host" , "VARCHAR(256)") -- possibly dnsname
, ("XML" , "TEXT NOT NULL") -- length 2^16
]

-- list of SqlValues. Order has to be the same as in the table field
-- definition above. Field types are checked implicitly by the
-- respective UsageRecord field types.
extractFields :: UsageRecord -> [SqlValue]
extractFields ur = [ toSql (urId ur)
                    , toSql (urCreateTime ur)
                    , toSql (uGlobal (urUser ur))]
  ++ map toSql -- all "maybe String"
    [ uLocal (urUser ur)
    , userVO (urUser ur)
    , jGlobal (urJob ur)
    , jLocal (urJob ur) ]
  ++ [ toSql (urStatus ur) -- restricted string
    , toSql (urStartTime ur) -- maybe utcTime
    , toSql (urEndTime ur) -- maybe utcTime
    , toSql (urWallDuration ur) -- maybe double, seconds
    , toSql (urCPUDuration ur) -- maybe double, seconds
    , toSql (urCharge ur) -- maybe double
    , toSql (urNodeCount ur) -- maybe double
    , toSql (urMachineName ur) -- maybe string
    , toSql (urQueue ur) -- maybe string
    , toSql (urHost ur) -- maybe string
    , toSql (urXml ur) -- string
    ]

-- the matching statement:
insertPrepStatement :: String
insertPrepStatement = "insert into " ++ table
  ++ "\n\t(" ++ sepBy ", " (map fst fields)
  ++ ") values ("
  ++ sepBy ", " (replicate (length fields) " ? ")
  ++ ");\n"

-- for building the statement:
table :: String
table = "JobUsageRecord"

-- a helper function: yield a single string where the list elements
-- are separated by the string given as the first argument.
sepBy :: String -> [String] -> String
sepBy sep strs | null strs = ""
               | otherwise = foldl1 (\a b -> a ++ sep ++ b) strs
--sepBy _ [] = ""
--sepBy _ [x] = x
--sepBy c (x:rest) = x ++ c : sepBy c rest

```

## Data Definition Statements

As we need to fix the table format anyway, we can also yield the matching table definition when desirable.

---

```
compiled code
-- statement to initialise the table:
initTableStatement :: String
initTableStatement = "CREATE TABLE " ++ table ++ "\n\t("
                    ++ sepBy "\n\t, "
                        ([ name ++ " " ++ typ
                          | (name,typ) <- fields ]
                        ++ indexOpts
                        -- ++ constraints
                        )
                    ++ ");\n"

-- index and key options for table creation
indexOpts = [ "PRIMARY KEY (RecordId)"
             , "INDEX time(CreateTime)"
             , "INDEX user (GlobalUserName)"
             , "INDEX exectime(StartTime,EndTime)"
             ]

-- could add constraints here... most of them in fields definition.
constraints = [""]
```

---

It would be nice to record versioned changes to the schema by concatenating DDL statements which capture the changes. However, this requires to select the matching version and only execute the new changes. And all would become unreadable. As we have just one live system...For now, manual work!

## Decoding Usage Record XML

Parsing XML data into the Usage record data structure is a separable task. Important, however: the Maybe type for missing fields.

We usually decode the data inside `JobUsageRecord`, but can first break up a single list node `UsageRecords` into its child elements.

---

```
compiled code
splitURs :: Element -> [Element]
splitURs e | qName(elName e) == "UsageRecords"
          = filter isUR (elChildren e)
          | isUR e      = [e] -- single record
          | otherwise   = error "unexpected XML content (no UsageRecords)"

isUR :: Element -> Bool
isUR e = qName (elName e) == "JobUsageRecord"

decodeUR :: Element -> UsageRecord
decodeUR uRecord
  | qName (elName uRecord) /= "JobUsageRecord"
    = error ("unexpected content: " ++ show uRecord)
  | noRecId = error "no recordID found"
  | otherwise = UsageRecord { urId = fromJust mId
                           , urCreateTime = mTime
                           , urStatus = status, urUser = user
                           , urJob = job
                           , urStartTime = start, urEndTime = end
                           , urWallDuration = walltime
                           , urCPUDuration = cputime
                           , urCharge = charge, urNodeCount = nodes
                           }
```

---

```

        , urMachineName = machine
        , urQueue = queue, urHost = host
        , urXml = ppElement uRecord -- pretty-print XML
    }
where mRecIdElem = findChildWith "RecordIdentity" uRecord
recIdElem = fromJust mRecIdElem
idNamesp = elName recIdElem
mId = findAttr (idNamesp{qName="recordId"} ) recIdElem
noRecId = case (mRecIdElem, mId) of
    (Nothing,_) -> True
    (_,Nothing) -> True
    other       -> False
mTime = do s <- findAttr (idNamesp{qName="createTime"}) recIdElem
        parseT s
status = read (stringInside "Status" uRecord)
userNode = findChildWith "UserIdentity" uRecord
user = case userNode of
    Nothing -> User "unknown" Nothing Nothing
    Just u ->
        (User {uGlobal= stringInside "GlobalUserName" u
              ,uLocal = maybeStringInside "LocalUserId" u
              ,userVO = Nothing -- TODO!
              })
jobNode = findChildWith "JobIdentity" uRecord
job = case jobNode of
    Nothing -> Job {jGlobal=Nothing,jLocal=Nothing}
    Just that ->
        (Job {jGlobal= maybeStringInside "GlobalJobId" that
              ,jLocal = maybeStringInside "LocalJobId" that})
start = do s <- maybeStringInside "StartTime" uRecord
        parseT s -- Maybe monad
end = do s <- maybeStringInside "EndTime" uRecord
        parseT s -- Maybe monad
tStr = maybeStringInside "WallDuration" uRecord
walltime = case tStr of
    Nothing -> Nothing
    Just t -> Just (
        if isDigit (head t)
        then read t
        else if 'P' == head t
        then duration2Double t
        else error ("not a duration: " ++ t))
cputime = Nothing -- TODO (same code as above)
charge = case (maybeStringInside "Charge" uRecord) of
    Nothing -> Nothing
    Just s -> Just (read s)
nodes = case (maybeStringInside "NodeCount" uRecord) of
    Nothing -> Nothing
    Just s -> Just (read s)
machine = maybeStringInside "MachineName" uRecord
queue = maybeStringInside "Queue" uRecord
host = maybeStringInside "Host" uRecord
-- match child element name, ignoring namespace URI and prefix
findChildWith :: String -> Element -> Maybe Element
findChildWith s e = filterChildName ((s==) . qName) e
stringInside :: String -> Element -> String
stringInside s e = case findChildWith s e of
    Nothing -> ""
    Just n -> strContent n
maybeStringInside :: String -> Element -> Maybe String
maybeStringInside s e = do x <- findChildWith s e
    return (strContent x)

```

---

The types we are using require some additional helper functions:

- Decoding an XML TimeDuration into fractional seconds:

---

```

duration2Double :: String -> Double
duration2Double ('P':dString)
  = let dayExtractFs = map getPart "YMD"
      (ymdDbl,rest) = runState (sequence dayExtractFs) dString
      timeStr = if null rest || head rest /= 'T'
                  then []
                  else tail rest
      timeExtractFs = map getPart "HMS"
      (hmsDbl,r2) = runState (sequence timeExtractFs) timeStr
      [y,m,d] = ymdDbl
      [h,min] = init hmsDbl
      (sec,parts) = properFraction (hmsDbl!!2)
      monthyeardays = -- compute from y and m... timeDuration is
                       -- not well specified, so we approximate a
                       -- month by 30 days :(
                       y * 365 + m * 30
      in foldl sumTimes monthyeardays
         (zip [d,h,min,fromIntegral sec] [1,24,60,60])
         + parts
sumTimes :: Double -> (Double,Double) -> Double
sumTimes y (n,f) = f*y+n

getPart :: Char -> State String Double
getPart c = do input <- get
  let (num,rest) = span digitOrDecimal input
  digitOrDecimal c = elem c "0123456789."
  if null rest
  then return 0 -- not found, empty input
  else if head rest == c
        then do put (tail rest)
                 return (read num)
        else do put input
                 return 0

```

---

This is actually not good; we put some effort into encoding a simpler format into this format before, now we decode it. The `arc-ur-logger` does not use `TimeDuration`, but merely fractional seconds (which, however, does not conform to the usage record format).

- The `UTCTime` data type we use for the time stamp does not have a conversion to `SqlData` (the older deprecated Haskell-98 time library is allegedly supported, but `CalendarTime` is converted into `SqlEpochTime`, which creates an SQL NULL value). We create an instance of our own.

---

```

instance SqlType UTCTime
  -- in fact, the driver returns us a string!
  where fromSql (SqlString s) = case parseTime defaultTimeLocale
                                   dbTimeFormat s of
                                   Nothing -> defaultDate
                                   Just utc -> utc
      toSql utc = SqlString (formatTime defaultTimeLocale dbTimeFormat utc)
dbTimeFormat :: String
dbTimeFormat = "%Y-%m-%d %H:%M:%S"
-- -- we are losing the picoseconds (SQL unable to set them)!

defaultDate :: UTCTime -- 1970-01-01 00:00:01
defaultDate = UTCTime (fromGregorian 1970 1 1) 1

-- other instance, like the older CalendarTime (does not work):
--   where fromSql (SqlEpochTime s) = readTime defaultTimeLocale "%s"

```

---

---

```

--                                ( show s )
--      toSql utc = SqlEpochTime (
--                                read (formatTime defaultTimeLocale "%s" utc))

-- another helper: parser function for time in ISO format.
parseT :: String -> Maybe UTCTime
parseT t = parseTime defaultTimeLocale "%Y-%m-%dT%H:%M:%S%QZ" t

```

---

## Database access

The database business is simply using the helper functions we have put together above. Care must be taken to catch SQL errors and not abort the whole statement sequence on a single error.

Open issue: How to get reasonable error details from HDBC.

---

```

-- compiled code
insertAllInto :: IConnection conn => conn -> [UsageRecord] -> IO ()
insertAllInto db urs = do let urVars = map extractFields urs::[[SqlValue]]
                          prep <- prepare db insertPrepStatement
                                -- ? marks for variables
                          sequence_ (map (executeWithCatch prep) urVars)
                          commit db

#ifdef DEBUG
  where prepare db stmt = -- compose a function to print statement
                        return (\s -> (putStrLn stmt >>
                                      putStrLn (show s)))
  commit db = putStrLn ("### commit " ++ db ++ " ###")
#endif

executeWithCatch stmt vars = E.catch (execute stmt vars >> return ())
                                   (\e -> putStrLn ("SQL error: "
                                                    ++ show e))

#ifdef DEBUG
  where execute prep var = prep var -- print it out only
#endif

-- test Usage Record parser (for debugging)
parseTest :: FilePath -> IO ()
parseTest file = do hSetBuffering stdout NoBuffering
                   hSetBuffering stderr NoBuffering
                   urFile <- readFile file
                   let records = parseXMLDoc urFile
                       toTest | isUR (fromJust records) = [fromJust records]
                              | otherwise = splitURs (fromJust records)
                   mapM_ parse toTest

  where parse ur = E.catch (let res = decodeUR ur
                           in putChar (evalUR res))
                        (\e -> do putStrLn ("\nfailed on " ++ show ur)
                               putStrLn (show e)
                           )

  -- rnf instance, hand-coded
  evalUR ur = foldr seq ' '
    [ urId ur, show (urCreateTime ur), show (urStatus ur)
    , show (urUser ur), show (urJob ur)
    , show (urStartTime ur), show (urEndTime ur)
    , show (urWallDuration ur), show (urCPUDuration ur)
    , show (urCharge ur), show (urNodeCount ur)
    , show (urMachineName ur), show (urQueue ur)
    , show (urHost ur), urXml ur]

```

---



About the database connection proper: we are using `connectODBC :: String -> IO Connection` from the ODBC driver. Documentation sez:

For information on the meaning of the passed string, please see:

[http://msdn2.microsoft.com/en-us/library/ms715433\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/ms715433(VS.85).aspx)

Thank you... this is the grammar for the connection string:

---

```
connection-string ::= empty-string[;]
                    | attribute[;]
                    | attribute; connection-string
empty-string ::=
attribute ::= attribute-keyword=attribute-value
            | DRIVER=[{]attribute-value[}]
attribute-keyword ::= DSN | UID | PWD
                  | driver-defined-attribute-keyword
attribute-value ::= character-string
driver-defined-attribute-keyword ::= identifier
```

---

And as we found out on a couple of howto pages online: The DSN is the data source name, so should be configured on the machine as connecting to the server and database. User and password (UID,PWD) are clear. The whole string needs a trailing semicolon!

---

*compiled code*

---

```
databaseODBC :: String
databaseODBC = sepBy ";" [ "DSN=nr4-usagerecords"
                          , "UID=sqluser", "PWD=sqluser"
                          , ";" ] -- trailing semicolon needed :P
```

---

## Using the database for agglomeration

We are most likely going to implement these parts in python later. Just for fun, we can select some agglomerated data from the tables:

---

*compiled code*

---

```
userSelectStatement :: String
userSelectStatement = unlines
  ["select GlobalUserName, count(*), sum(Charge), "
  , "      month(StartTime), year(StartTime)"
  , "from " ++ table
  , "  where GlobalUserName = ? and StartTime is not NULL"
  , "group by month(StartTime), year(StartTime)"
  ]

statsUser :: IConnection conn => conn -> String -> IO ()
statsUser db username
  = E.catch getStats (\e -> putStrLn ("SQL error: " ++ show e))
  where display :: [SqlValue] -> IO ()
        display [_ , jobCountS, chargeSumS, monthS, yearS]
          = do putStrLn
              (show ((fromSql monthS, fromSql yearS)::(Int,Int))
                ++ ": Charge " ++ show (fromSql chargeSumS::Double)
                ++ " in " ++ show (fromSql jobCountS::Int) ++ " jobs"
              )
        getStats = do results <- quickQuery db
                      userSelectStatement [toSql username]
                      putStrLn (username ++
```

```

                                "\n" ++ replicate (length username) '-'')
mapM_ display results
putStrLn "-----"

```

---

This code does a select of all data for one particular GlobalUserName, grouping usage data (job count and charge sum) by month, and year.

---

```

                                compiled code
stats :: IO ()
stats = do db <- connectODBC databaseODBC
        E.catch
            (do users <- quickQuery db
              ("select distinct GlobalUserName from "
               ++ table)
              []
              mapM_ (statsUser db . fromSql . head) users
            ) (\e -> do putStrLn ("Sql error: " ++ show e))
        disconnect db

```

---

## Furthermore...

It would be nice to extend the schema to more than one single table. Several improvements come into mind...

- Separate user table, foreign key links to JobUsageRecords The table could be filled by a trigger:

---

```

CREATE TRIGGER newUser
BEFORE INSERT ON JobUsageRecord
FOR EACH ROW BEGIN
    IF NOT (NEW.GlobalUserName IN
            (SELECT GridUser.GlobalUserName FROM GridUser))
    THEN INSERT INTO GridUser(GlobalUserName , LocalUserId)
            values (NEW.GlobalUserName, NEW.LocalUserId)
    END IF
END

```

---

Later on, we might store VO information as well, fill a table VO in quite the same way, and add a GridUser\_VO connection table as well. Performance will of course drop then.

- We should keep pre-agglomerated data for standard queries around to avoid repeatedly traversing the whole set.
- In the same spirit, saving away historic data into another table might pay off, accelerating the insertion and selection. Agglomerating queries have to be aware and split in two: current and historic.

All fairly standard...

## Deriving a specification

The usage record format specification includes many optional fields and extensions, and only defines recordID and status as mandatory – not enough to rely on for statistics generation.

Since the SGAS project did not specify which fields are expected to be in their usage records, we derive a specification ourselves from the de-facto standard.

### Usage record samples:

Generated by `arc-ur-logger`:<sup>2</sup>

```
<?xml version="1.0" encoding="UTF-8" ?>
<ur:JobUsageRecord xmlns:ur="http://schema.ogf.org/urf/2003/09/urf">
  <ur:RecordIdentity
    ur:createTime="2009-07-07T09:06:52Z"
    ur:recordId="gsiftp://benedict.grid.aau.dk:2811/jobs/2087912469575971260292766" />
  <ur:JobIdentity>
    <ur:GlobalJobId>
      gsiftp://benedict.grid.aau.dk:2811/jobs/2087912469575971260292766
    </ur:GlobalJobId>
    <ur:LocalJobId>437190.mother</ur:LocalJobId>
  </ur:JobIdentity>
  <ur:UserIdentity>
    <ur:LocalUserId>grid</ur:LocalUserId>
    <ur:GlobalUserName>
      /O=Grid/O=NorduGrid/OU=ndgf.org/CN=Henrik Thostrup Jensen
    </ur:GlobalUserName>
    <vo:VO vo:type="voms"
      xmlns:vo="http://www.sgas.se/namespaces/2009/05/ur/vo">
      <vo:Name>bio.ndgf.org</vo:Name>
      <vo:Issuer>/O=Grid/O=NorduGrid/CN=host/voms.ndgf.org</vo:Issuer>
      <vo:Attribute>
        <vo:Group>bio.ndgf.org/test</vo:Group>
        <vo:Role>VO-dev</vo:Role>
      </vo:Attribute>
      <vo:Attribute>
        <vo:Group>bio.ndgf.org</vo:Group>
      </vo:Attribute>
      <vo:Attribute>
        <vo:Group>bio.ndgf.org/test</vo:Group>
      </vo:Attribute>
    </vo:VO>
  </ur:UserIdentity>
  <ur:JobName>Sleepy test job</ur:JobName>
  <ur:Charge>1.0</ur:Charge>
  <ur:Status>completed</ur:Status>
  <ur:MachineName>benedict.grid.aau.dk</ur:MachineName>
  <ur:Queue>brother</ur:Queue>
  <ur:Host>brother16</ur:Host>
  <ur:NodeCount>1</ur:NodeCount>
  <deisa:SubmitTime
    xmlns:deisa="http://rmis.deisa.org/acct">
    2009-07-07T09:06:37Z
  </deisa:SubmitTime>
  <ur:StartTime>2009-07-07T09:06:37Z</ur:StartTime>
  <ur:EndTime>2009-07-07T09:06:52Z</ur:EndTime>
  <ur:WallDuration>PT1S</ur:WallDuration>
```

<sup>2</sup>obtained from Henrik Jensen on August 14 2009.

```
<ur:CpuDuration>PT0S</ur:CpuDuration>
</ur:JobUsageRecord>
```

### Imported from Monalisa:<sup>3</sup>

```
<ur:JobUsageRecord xmlns:ur='http://schema.ogf.org/urf/2003/09/urf' \
  xmlns:sgas='http://www.sgas.se/namespaces/2009/07/ur'>
  <ur:RecordIdentity ur:recordId='UiB_1247672281271' ur:createTime='2009-07-15T17:38:01Z' />
  <ur:JobIdentity>
    <ur:GlobalJobId>UiB_1247672281271</ur:GlobalJobId>
    <ur:LocalJobId>UiB_compute-3-31.local:8095</ur:LocalJobId>
  </ur:JobIdentity>
  <ur:UserIdentity>
    <ur:LocalUserId>aliprod</ur:LocalUserId>
    <ur:GlobalUserName>aliprod</ur:GlobalUserName>
    <vo:VO xmlns:vo='http://www.sgas.se/namespaces/2009/05/ur/vo' vo:type='monalisa'>
      <vo:Name>alice.cern.ch</vo:Name>
    </vo:VO>
  </ur:UserIdentity>
  <ur:Status>completed</ur:Status>
  <ur:StartTime>2009-07-15T16:13:13Z</ur:StartTime>
  <ur:NodeCount>1</ur:NodeCount>
  <ur:WallDuration>PT5088.0S</ur:WallDuration>
  <ur:CpuDuration>PT88.0S</ur:CpuDuration>
  <ur:MachineName>grid.bccs.uib.no</ur:MachineName>
  <ur:Host>compute-3-31.local</ur:Host>
  <sgas:KSI2KWallDuration>PT8488.4399999999999S</sgas:KSI2KWallDuration>
  <sgas:KSI2KCpuDuration>PT150.47999999999993S</sgas:KSI2KCpuDuration>
</ur:JobUsageRecord>
```

Missing here: special cases, especially jobs which failed or which were canceled. Henrik will perhaps provide more...

### Generated by MiG plugin (version: August 17):

```
<?xml version="1.0" ?>
<JobUsageRecord>
  <RecordIdentity createTime="2009-08-17T10:56:53.244404Z"
    recordId="15_7_21_2009__11_42_47_portal.grid.dk.0"/>
  <JobIdentity>
    <GlobalJobId>15_7_21_2009__11_42_47_portal.grid.dk.0</GlobalJobId>
    <LocalJobId>210709-141541.15603</LocalJobId>
  </JobIdentity>
  <UserIdentity>
    <GlobalUserName>
      /C=DK/ST=NA/L=NA/O=DIKU/OU=NA/CN=Jonas Bardino/emailAddress=bardino@diku.dk
    </GlobalUserName>
    <LocalUserId>mig-griddk</LocalUserId>
  </UserIdentity>
  <Charge formula="nodes * wall_duration(sec)">38.0</Charge>
  <Status>completed</Status>
  <MachineName>benedict.grid.aau.dk.1</MachineName>
  <NodeCount>1</NodeCount>
  <Host>brother1</Host>
  <StartTime>2009-07-21T12:17:09Z</StartTime>
  <EndTime>2009-07-21T12:17:47Z</EndTime>
  <WallDuration>PODT00H00M38.000000S</WallDuration>
</JobUsageRecord>
```

<sup>3</sup>obtained from Henrik Jensen on August 14 2009.

### MiG plugin, failed job (resource error, job timed out):

```
<?xml version="1.0" ?>
<JobUsageRecord>
  <RecordIdentity createTime="2009-08-17T10:56:47.996227Z"
    recordId="14_7_21_2009__9_56_43_portal.grid.dk.0"/>
  <JobIdentity>
    <GlobalJobId>14_7_21_2009__9_56_43_portal.grid.dk.0</GlobalJobId>
    <LocalJobId>210709-125530.5341</LocalJobId>
  </JobIdentity>
  <UserIdentity>
    <GlobalUserName>
      /C=DK/ST=NA/L=NA/O=DIKU/OU=NA/CN=Jonas Bardino/emailAddress=bardino@diku.dk
    </GlobalUserName>
    <LocalUserId>mig-griddk</LocalUserId>
  </UserIdentity>
  <Status>failed</Status>
  <MachineName>benedict.grid.aau.dk.0</MachineName>
  <NodeCount>1</NodeCount>
  <Host>sister1</Host>
  <StartTime>2009-07-21T10:56:58Z</StartTime>
</JobUsageRecord>
```

### MiG plugin, aborted job (canceled by user):

```
<?xml version="1.0" ?>
<JobUsageRecord>
  <RecordIdentity createTime="2009-08-17T13:04:48.970143Z"
    recordId="3_7_13_2009__14_56_38_portal.grid.dk.0"/>
  <JobIdentity>
    <GlobalJobId>3_7_13_2009__14_56_38_portal.grid.dk.0</GlobalJobId>
    <LocalJobId>130709-165706.29415</LocalJobId>
  </JobIdentity>
  <UserIdentity>
    <GlobalUserName>/O=Grid/O=NorduGrid/OU=grid.dk/CN=Jost Berthold</GlobalUserName>
    <LocalUserId>mig-griddk</LocalUserId>
  </UserIdentity>
  <Status>aborted</Status>
  <MachineName>benedict.grid.aau.dk.0</MachineName>
  <NodeCount>1</NodeCount>
  <Host>sister2</Host>
  <StartTime>2009-07-13T14:58:29Z</StartTime>
</JobUsageRecord>
```

### job canceled before starting (+older mRSL version:)

```
<?xml version="1.0" ?>
<JobUsageRecord>
  <RecordIdentity createTime="2009-08-17T12:30:43.966214Z"
    recordId="15999_6_3_2007__17_46_3_mig-1.imada.sdu.dk.0"/>
  <JobIdentity>
    <GlobalJobId>15999_6_3_2007__17_46_3_mig-1.imada.sdu.dk.0</GlobalJobId>
  </JobIdentity>
  <UserIdentity>
    <GlobalUserName>Alex_Esmann</GlobalUserName>
  </UserIdentity>
  <Status>aborted</Status>
  <NodeCount>1</NodeCount>
  <ProjectName>NQueen_Test</ProjectName>
</JobUsageRecord>
```

## Common specification derived from examples:

Field name	Classification	Description
RecordIdentity	mandatory	containing <code>createTime</code> and <code>recordId</code>
JobIdentity	optional	present in all examples. However, not required.
UserIdentity	mandatory	we should require at least <code>GlobalUserName</code>
VO	optional	VO part of a user identity, can have different shape, or be missing. Could reduce format to a set of URLs (host/group/role).
Status	mandatory	Should be one of “failed”, “aborted” or “completed”.
StartTime	optional	<b>mandatory</b> if status completed/failed.
EndTime	optional	<b>mandatory</b> if status completed. Walltime/Charge can be computed from it.
NodeCount	optional	(default 1 for computing charge?)
WallDuration	optional	measured time on host, <b>otherwise computed</b> as end-start (in MiG).
CpuDuration	optional	measured on the host
Charge	optional	used in MiG and arc-ur-logger to provide a rough measure for the CPU resource usage (walltime * nodecount). <sup>a</sup>
JobName	optional	user-specified
ProjectName	optional	user-specified
MachineName	optional	executing cluster
Queue	optional	only generated by ARC/ only for LRMS-based systems
Host	optional	the executing node. BTW: should we reveal this information at all?

<sup>a</sup>Monalisa seems to have a different measure “KSI2K”, timing values scaled according to a performance factor. (Right?)