GRACeFUL WP4 - DSL

Domain-Specific Language for System Dynamics Models

Patrik Jansson, Maximilian Algehed, Sólrún Einarsdóttir, and *Alex Gerdes*

11 December 2017

Chalmers University of Technology

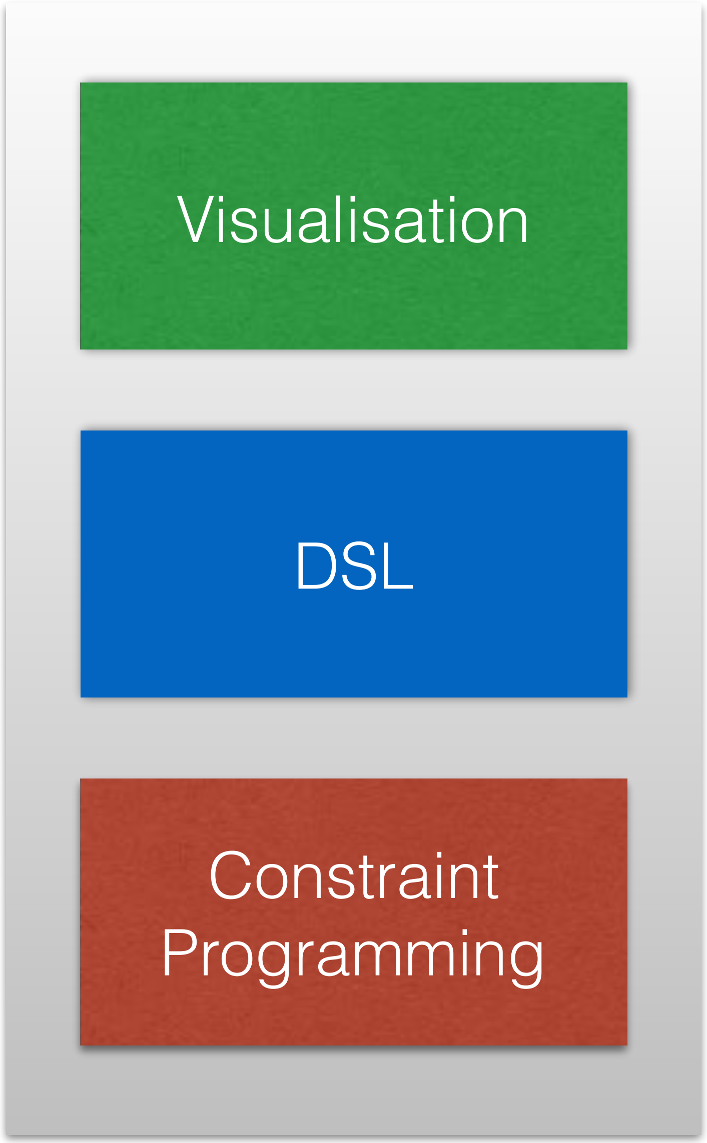
Introduction

- What is the purpose of GRACeFUL?
- What is a GRACeFUL Concept Map?
- Use Concept Maps and Constraint Programming instead of simulation

Domain specific language

- Use a DSL to express Concept Maps

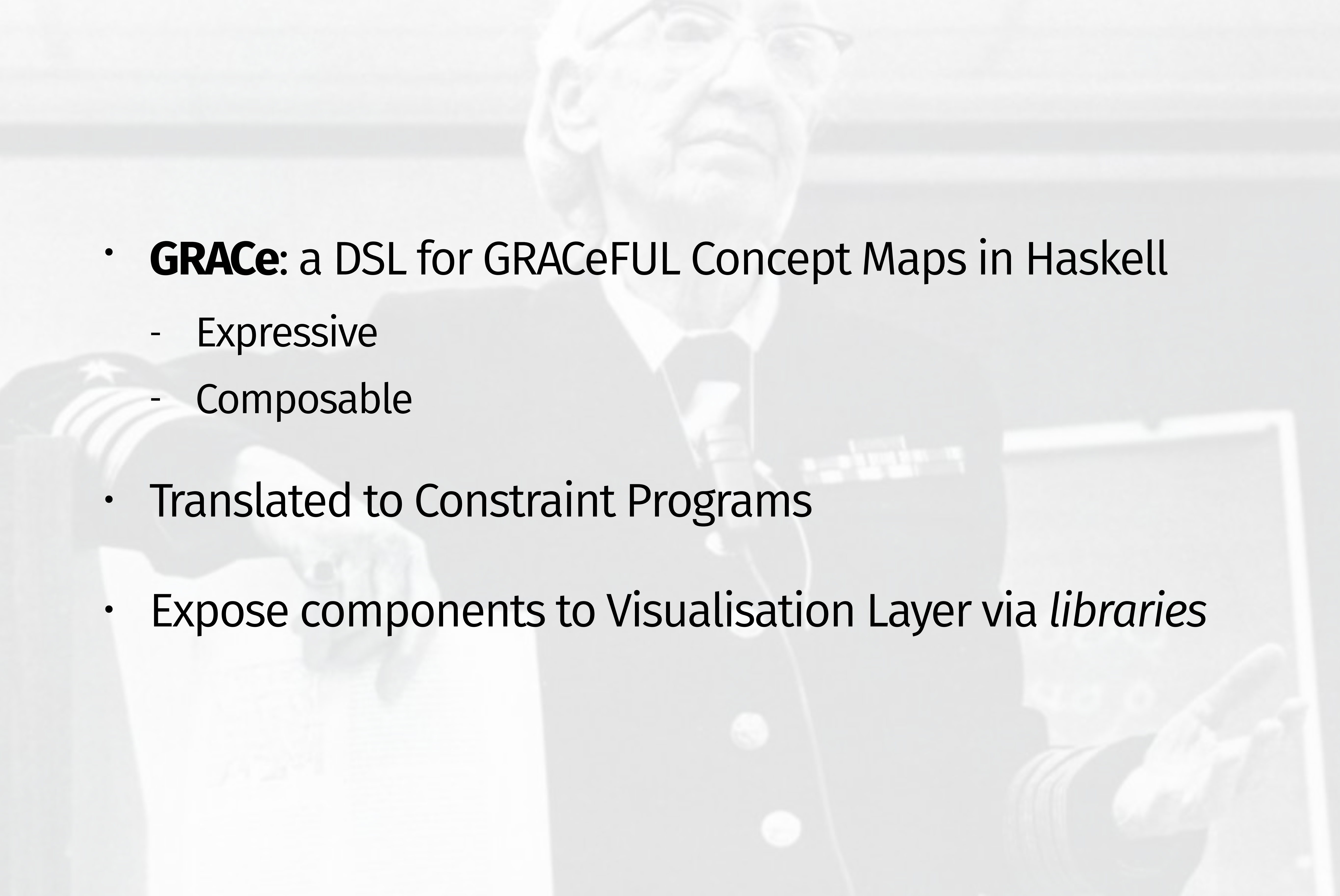
- Why use a DSL (and not directly translate to CP)?
  - Simple interface for Visualisation Layer
  - Abstract away from Constraint Programming layer
  - Validate correctness of generated model



Haskell

- We have implemented the DSL in *Haskell*
- Haskell serves as the *host language*
- Haskell is a functional programming language
  - Algebraic datatypes
  - Strongly typed
  - Good compiler: GHC

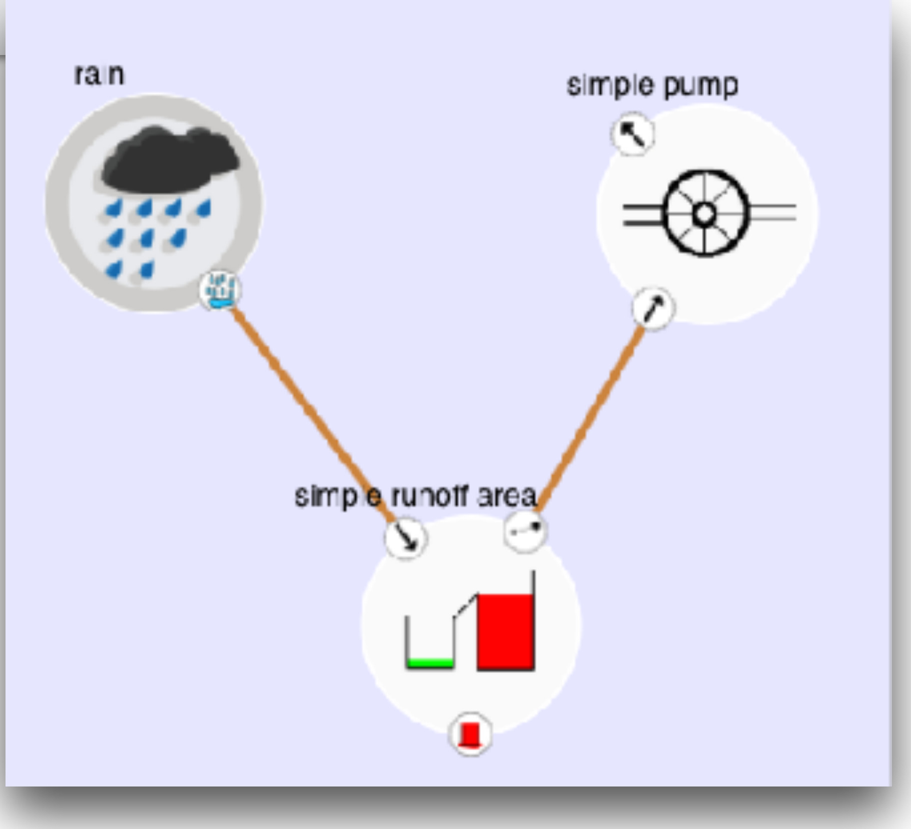
GRACe DSL



- **GRACe**: a DSL for GRACeFUL Concept Maps in Haskell
  - Expressive
  - Composable
- Translated to Constraint Programs
- Expose components to Visualisation Layer via *libraries*

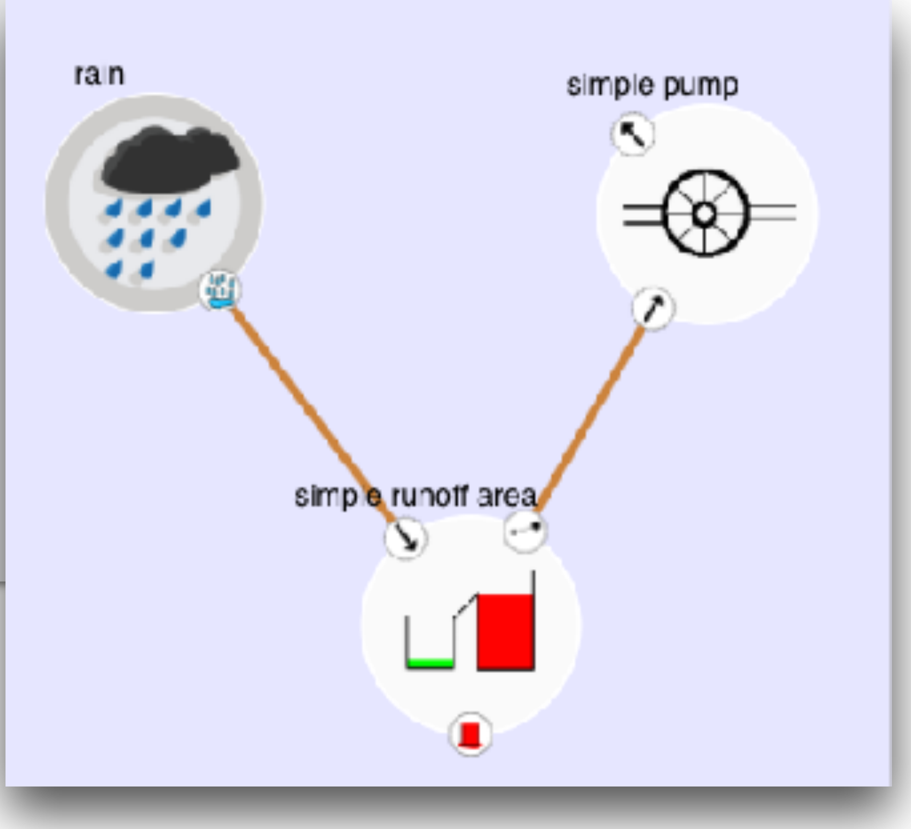
An example

```haskell
example :: GCM ()
example = do
  (inflowP, outflowP) <- pump 5
  (inflowS, outletS, overflowS) <- runoffArea 5
  rainflow <- rain 10

  link inflowP outletS
  link inflowS rainflow

  output overflowS "Overflow"
```



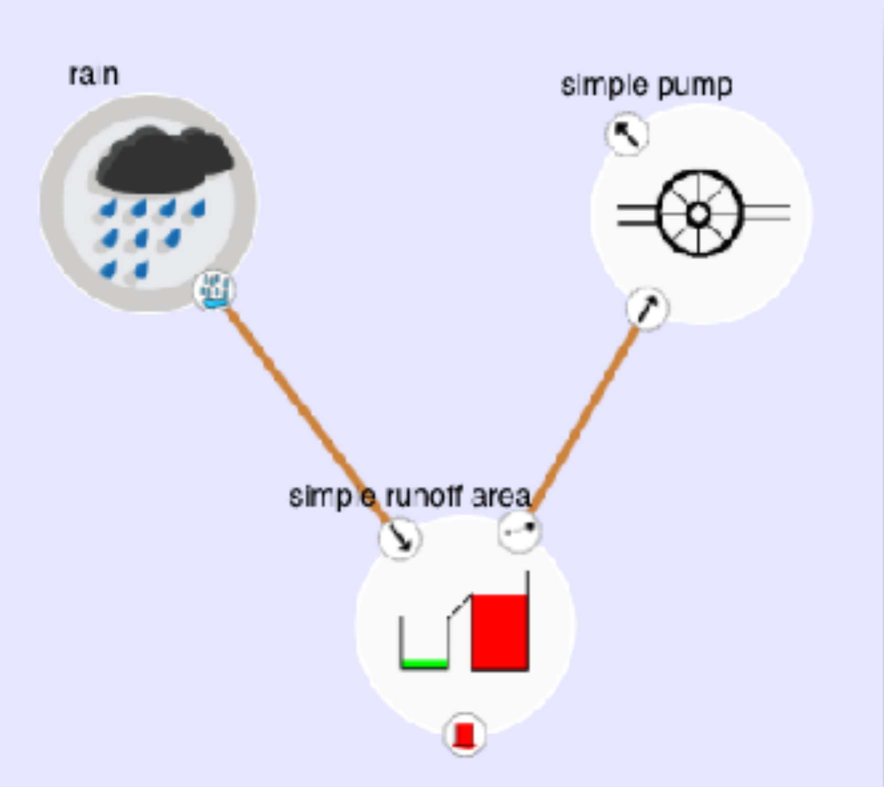
An example

```haskell
rain :: Float -> GCM (Port Float)
rain amount = do
  port <- createPort
  set port amount
  return port

pump :: Float -> GCM (Port Float, Port Float)
pump maxCap = do
  inPort  <- createPort
  outPort <- createPort

  constrain $ do
    inflow  <- value inPort
    outflow <- value outPort

    assert $ inflow === outflow
    assert $ inflow `inRange` (0, maxCap)

  return (inPort, outPort)
```



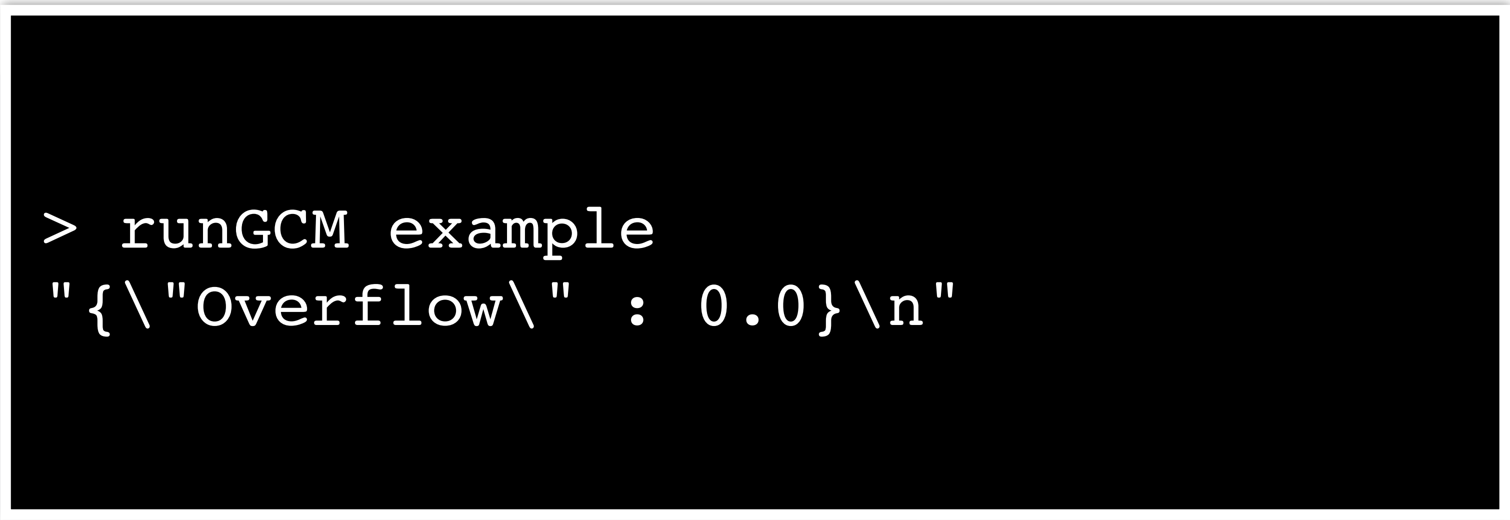
An example

```
runoffArea :: Float -> GCM (Port Float, Port Float, Port Float)
runoffArea cap = do
  inflow <- createPort
  outlet <- createPort
  overflow <- createPort

  constrain $ do
    currentStored <- createLVar

    inf <- value inflow
    out <- value outlet
    ovf <- value overflow
    sto <- value currentStored

    assert $ sto === inf - out - ovf
    assert $ sto `inRange` (0, cap)
    assert $ (ovf .> 0) ==> (sto === cap)
    assert $ ovf .>= 0

  return (inflow, outlet, overflow)
```



An example

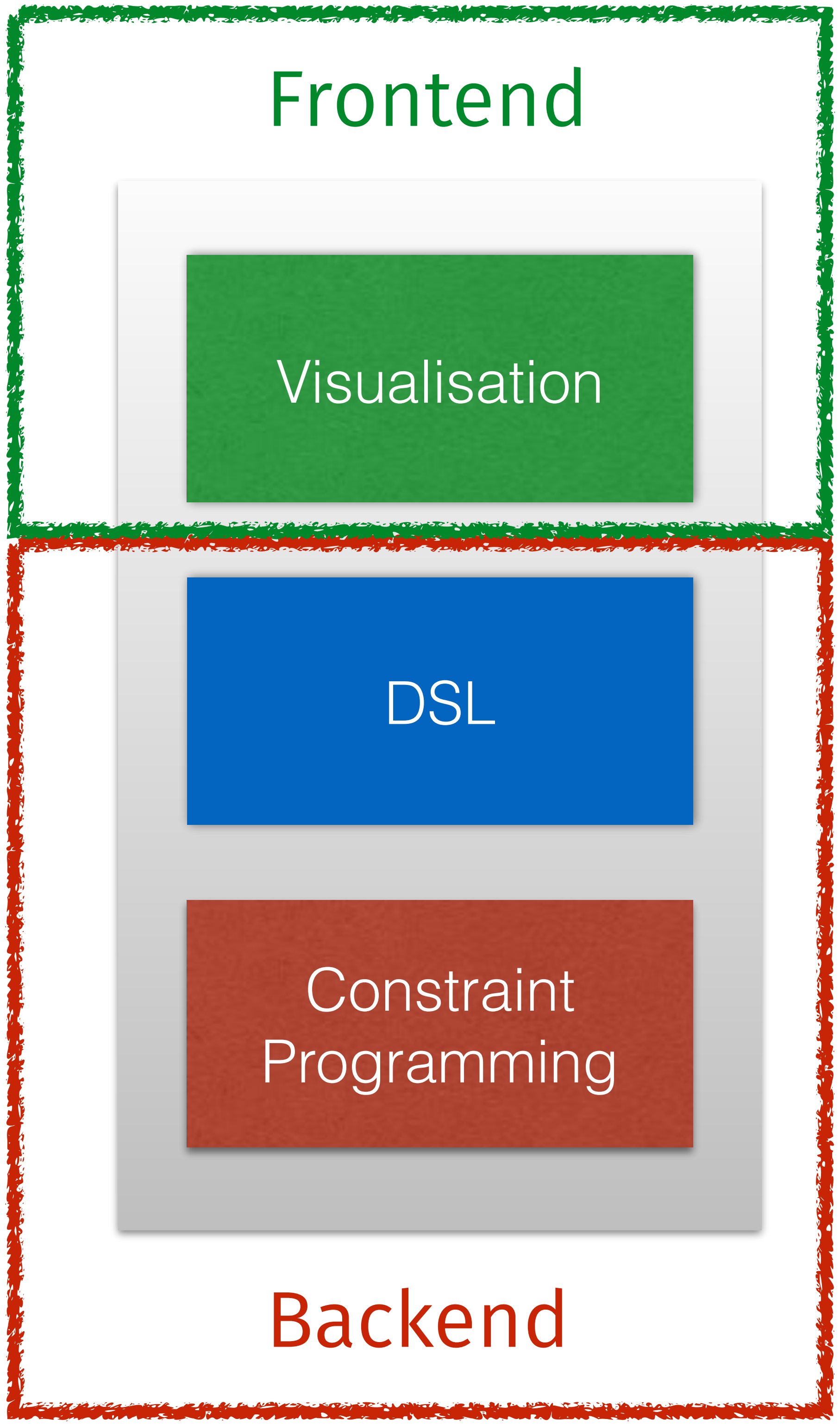
```
var float: v6;
var float: v5;
var float: v4;
var float: v3;
var float: v2;
var float: v1;
var float: v0;

constraint ((v0) == (v1));
constraint (((0.0) <= (v0)) /\ ((v0) <= (5.0)));
constraint ((v5) == (((v2) - (v3)) - (v4)));
constraint (((0.0) <= (v5)) /\ ((v5) <= (5.0)));
constraint ((not (((not ((not ((0.0) < (v4)))))) /\
           ((not ((v5) == (5.0))))))));
constraint ((0.0) <= (v4));
constraint ((v6) == (10.0));
constraint ((v0) == (v3));
constraint ((v2) == (v6));

solve satisfy;
output ["{\"Overflow\" : \(v4)}"]
```



```
> runGCM example
"{\"Overflow\" : 0.0}\n"
```

GRACeServer

- Expose GRACe DSL via webservices:
  - Query libraries
  - Evaluate DSL programs
  - Exchange format in JSON
- We use the Servant library
- Deployed via Docker
- Typed -> untyped -> typed



TypedValues

```
data TypedValue where
  (:::) :: a -> Type a -> TypedValue

data Type a where
  Int          :: Type Int
  Float        :: Type Float
  Port         :: Type a -> Type (Port a)
  GCM          :: Type a -> Type (GCM a)
  List         :: Type a -> Type [a]
  Tag          :: String -> Type a -> Type a
  (:->)        :: Type a -> Type b -> Type (a -> b)
  . . .
```

- Combine value with its type representation
- Hide type using existential quantification
- Tags for annotating meta data (such as names and images)

Example library

```
library :: Library
library = Library "crud"
    [ item "rain" $
        rain ::: "amount" #
          tFloat .-> tGCM             (tPort $ "rainfall" # tFloat)
    , item "pump" $
        pump ::: "capacity" #
          tFloat .-> tGCM (tPair      (tPort $ "inflow"   # tFloat)
                                      (tPort $ "outflow"  # tFloat))
    , item "runoff area" $
        runoffArea ::: "storage capacity" #
          tFloat .-> tGCM (tTuple3 (tPort $ "inflow"   # tFloat)
                                   (tPort $ "outlet"   # tFloat)
                                   (tPort $ "overflow" # tFloat))
    ]
```

Conclusions

- We have constructed a *DSL* for GRACeFUL Concept Maps
- We can define *libraries* of components
- Using the *GRACeServer* the Visualisation Layer can construct and evaluate GRACeFUL Concept Maps