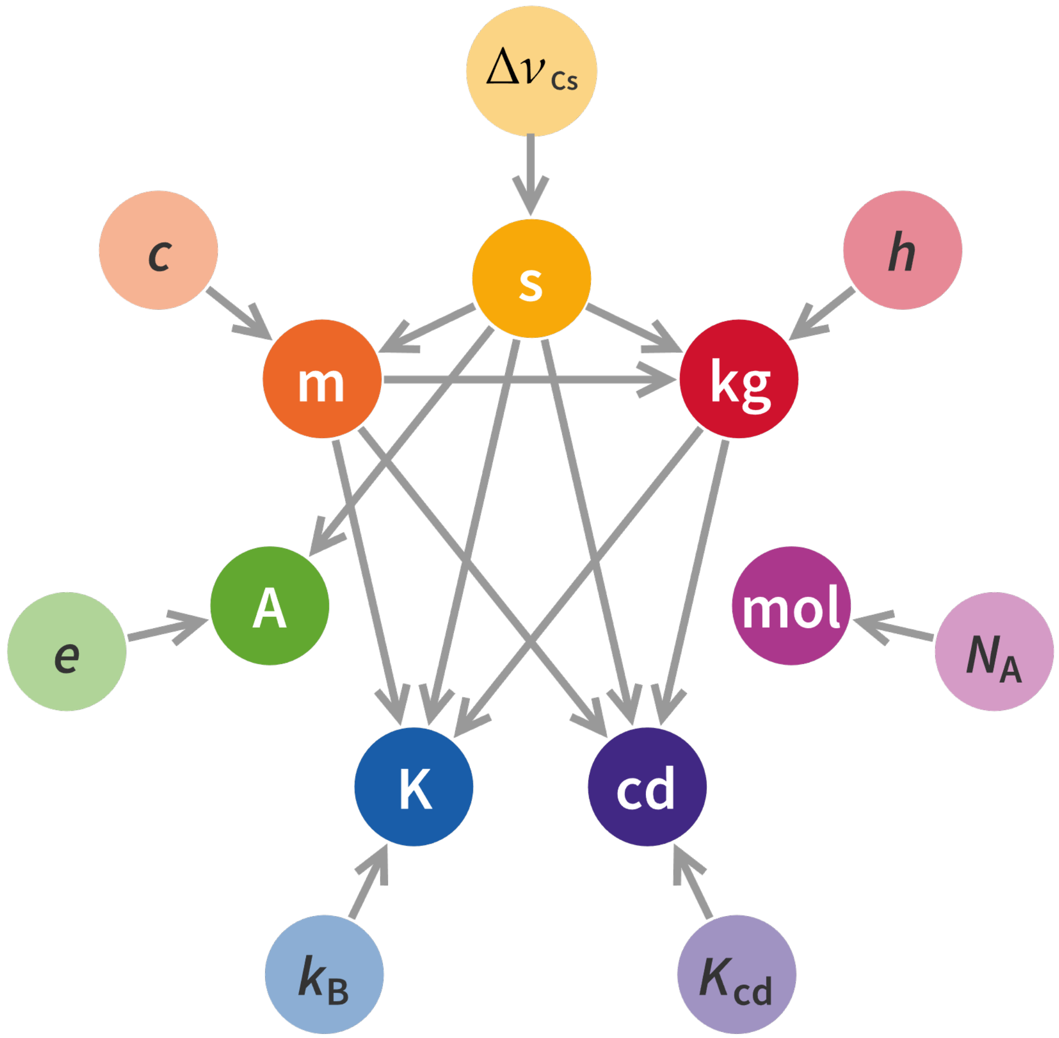
International System of Quantities library in VDM

The 21st Overture Workshop

10th March 2023

Leo Freitas
School of Computing, Newcastle University



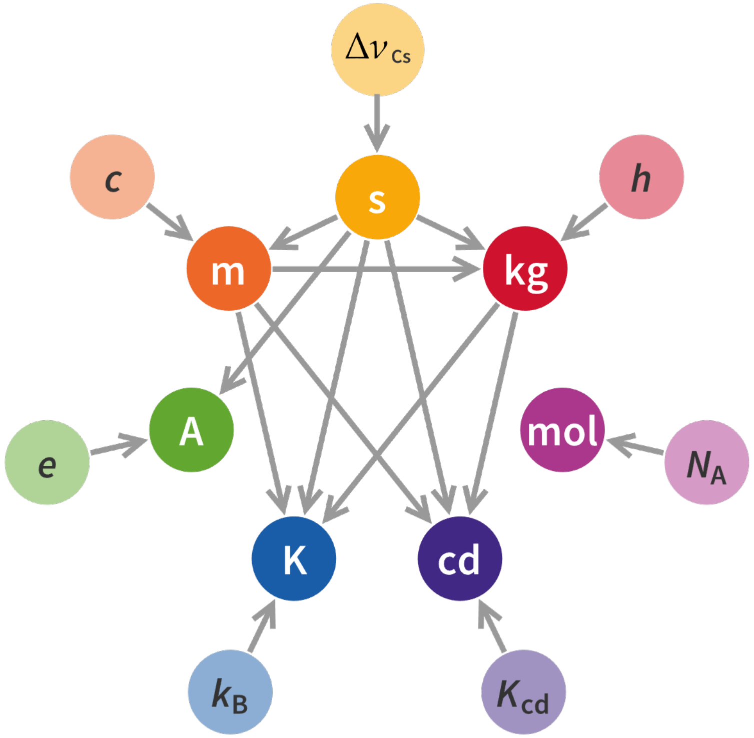
Introduction

- Most used system of measurements across multiple disciplines
- Elegant, minimal and coherent
- Seven base units, and new user units are coherent derivations
- Formal representation of these units is important
- Reduced error-prone and tedious calculations
- History: used for personalised medicine application conversions

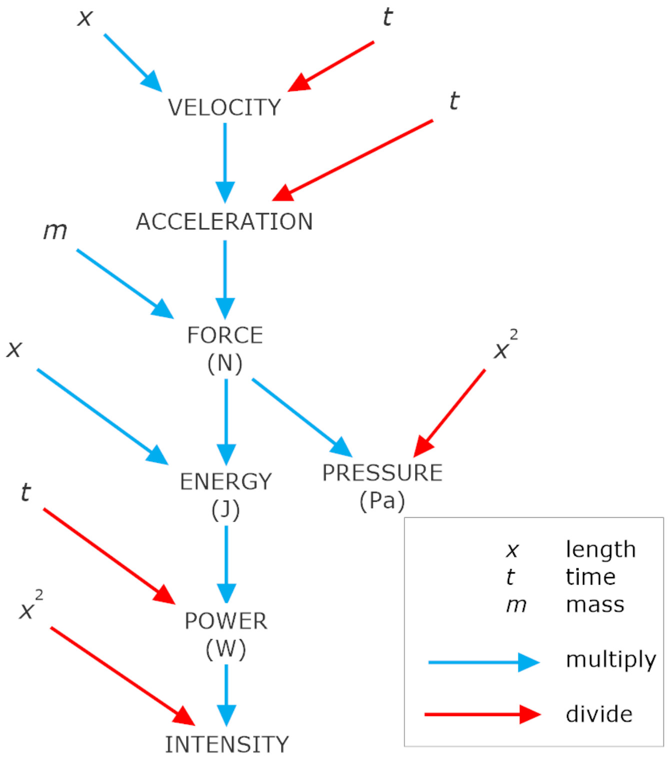
Base Units

Quantity	Symbol	Dimension	SI Name	SI Symbol
Length	l	L	metre	**M**
Mass	m	M	kilogram	**kg**
Time	t	T	second	**s**
Electric Current	I	I	ampere	**A**
Thermodynamic Temperature	T	Θ	kelvin	**K**
Amount of Substance	n	N	mole	**mol**
Luminous Intensity	I_v	J	candela	**cd**

Derived Units



ISQ Base Unit Relationships



ISQ Unit Combination and Conversion

Dimension Vector

- Map of each quantity dimension (or unit) in terms of its relations with other quantity base units **(1)**
- Enables representation of derived units through base unit conversions (e.g. 1km = 1000m)
- Most common dimensions (7 base and 15 derived) are predefined
- Dimensionless vectors (dimension mapped to zero) define pure quantities (e.g. radians) **(2)**
- Operators exist to combine and manipulate (e.g. multiply, invert, etc.) derived dimensions **(3)**

1

```
DimensionVector = DimensionlessVector
inv dv ==
    --OnFail(4060, "Dimension vector has only dimensionless (0) ranges for %1s", dom(dv :> {0}))
    (rng(dv :-> {0}) <> {});
```

2

```
DimensionVector0= map Dimension to int;
Leo Freitas, 3 m
DimensionlessVector = DimensionVector0
inv dv ==
    --@doc needs info on all known dimensions
    --@OnFail(4060, "Dimensionless vector missing %1s dimensions = %2s", DIMENSIONS \ dom dv, dv)
    (dom dv = DIMENSIONS)
    and
    --@doc needs at least one dimension set
    true--rng(dv :-> {0}) <> {}
    --(dunion rng dv) \ {0} <> {}
    ;
```

3

```
dim_comp: DimensionlessVector * DimensionlessVector -> DimensionlessVector
dim_comp(d1, d2) == { d |-> d1(d) + d2(d) | d in set dom d1 inter dom d2 }

dim_inv: DimensionlessVector -> DimensionlessVector
dim_inv(di) == { d |-> -di(d) | d in set dom di }

dim_div: DimensionlessVector * DimensionlessVector -> DimensionlessVector
dim_div(d1, d2) == dim_comp(d1, dim_inv(d2))
```

Quantities

- Represents magnitude of a dimension
- Used for conversion between different measurement systems (SI x BSI)
- Comparison and ordering within same dimension
- Quantities are `real` typed
- Different varieties defined **(1)**
    - Integer magnitudes
    - Single dimension quantities
- Several operators for quantities **(2)**
    - Multiplication
    - Subtraction
    - Replication
    - Etc.

```
Quantity ::
    mag: Magnitude
    dim:- DimensionlessVector
--eq  mk_Quantity(m1, d1) = mk_Quantity(m2, d2) == m1 = m2 --and d1 = d2
ord mk_Quantity(m1, -) < mk_Quantity(m2, -) == m1 < m2 ; --and d1 = d2;

--@doc non-zero quantity in any dimension

QuantityN0 = Quantity
inv mk_Quantity(m, -) == is_MagnitudeN0(m);

--@doc single dimension quantity (e.g. 3km)

SQuantity = Quantity
inv mk_Quantity(-, d) == is_SingleDimension(d);

--@doc non-zero single dimension quantity

SQuantityN0 = SQuantity
inv sq == is_QuantityN0(sq);


IntQuantity = Quantity
inv iq == is_int(iq.mag);
```

1

```
quant_dim_eq: Quantity * Quantity -> boo
quant_dim_eq(mk_Quantity(-, d1), mk_Quan

--@doc multiplying quantities multiply t

quant_times: Quantity * Quantity -> Quantity
quant_times(mk_Quantity(m1, d1), mk_Quantity(m2, d2)) ==
    mk_Quantity(m1*m2, dim_comp(d1, d2));

--@doc useful to make metre^3

quant_itself_n: Quantity * nat1 -> Quantity
quant_itself_n(mk_Quantity(m, d), n) == mk_Quantity(m, dim_comp_n(d, n));

    --@doc dividing quantities divides their magnitude and divides their dimensions
--@doc notice the second argument must be a non-zero quantity

quant_div: Quantity * QuantityN0 -> Quantity
quant_div(mk_Quantity(m1, d1), mk_Quantity(m2, d2)) ==
    mk_Quantity(m1/m2, dim_div(d1, d2));

--@doc inverting quantities invert their magnitude and invert their dimensions

quant_inv: QuantityN0 -> Quantity
```

2

Measurement Systems

- Group of quantities with specific dimensions and conversion schemas **(1)**
- Measurement systems are defined for all dimension (base or derived)
- Conversion schemas define how conversion between dimensions of different measurement systems can be done **(2)**
- SI conversion schema = identity map
- BSI conversion schema **(3)**

1

```
MeasurementSystem ::
    quantity: Quantity
      schema: ConversionSchema
        unit: UnitSystem
inv mk_MeasurementSystem(mk_Quantity(-, d), s, -) == dom d = dom s
eq  mk_MeasurementSystem(q1, s1, u1)
    =
    mk_MeasurementSystem(q2, s2, u2)
    ==
    q1 = q2 and s1 = s2 and u1 = u2
ord mk_MeasurementSystem(mk_Quantity(m1, -), -, -)
    <
    mk_MeasurementSystem(mk_Quantity(m2, -), -, -)
    ==
    --OnFail(4087, "Cannot compare measurement systems: %1s(%2s) < %3s(%4s)?", u1, m1, u2, m2)
    (m1 < m2)-- and s1 = s2 and u1 = u2)
;
```

2

```
ConversionSchema = map Dimension to MagnitudeN0
inv cs == dom cs = DIMENSIONS;
```

3

```
--@doc British Imperial System; choose Rankine instead of Farenheit for offset simplicity
BIS: ConversionSchema = CONV_ID ++
    { <Length> |-> 0.9143993, <Mass> |-> 0.453592338, <Temperature> |-> 5/9 };
```

Scaling and Conversion

- Scaling is the product of magnitudes
- Scaling two quantities ignores dimension vectors, unlike scaling measurement systems
- Scaling takes dimension vector of leading entity (e.g. km/h * miles/h results in km/h)

- Converts magnitudes using the given conversion schema
- Quantity conversion uses set product of integer exponents for corresponding schemas, where zero dimensions vanish
- Be aware of potential real precision issues

```
scaleQ: Magnitude * Quantity -> Quantity
scaleQ(m1, mk_Quantity(m2, d)) == mk_Quantity(m1 * m2, d);

scaleMS: Magnitude * MeasurementSystem -> MeasurementSystem
scaleMS(m1, mk_MeasurementSystem(q, s, u)) == mk_MeasurementSystem(scaleQ(m1, q), s, u);
```

```
quant_conv: ConversionSchema * DimensionlessVector -> MagnitudeN0
quant_conv(cs, dv) ==
    prods_r({ cs(i)**dv(i) | i in set dom cs inter dom dv })
pre
    dom cs = dom dv;

ms_conv_eq: MeasurementSystem * MeasurementSystem -> bool
ms_conv_eq(m1, m2) == m1.schema = m2.schema and m1.unit = m2.unit;
```

Common Prefixes

- Prefix enable ease of (re)use
- Prefixes work on
  - Measurement systems
  - Quantity
  - Magnitude
- Uses VDM Union types to reduce duplication of code

```
mag: Prefix -> Magnitude
mag(x) ==
    cases true:
        (is_Magnitude(x))         -> x,
        (is_Quantity(x))          -> x.mag,
        (is_MeasurementSystem(x)) -> x.quantity.mag
    end;

Launch | Debug
scale_prefix: Prefix * Magnitude -> Prefix
scale_prefix(x, p) ==
    cases true:
        (is_Magnitude(x))         -> x * p,
        (is_Quantity(x))          -> scaleQ(p, x),
        (is_MeasurementSystem(x)) -> scaleMS(p, x)
    end;

Launch | Debug
deca: Prefix -> Prefix
deca(x) == scale_prefix(x, PREFIX_DECA );

Launch | Debug
hecto: Prefix -> Prefix
hecto(x)== scale_prefix(x, PREFIX_HECTO);

Launch | Debug
kilo: Prefix -> Prefix
kilo(x) == scale_prefix(x, PREFIX_KILO );

Launch | Debug
mega: Prefix -> Prefix
mega(x) == scale_prefix(x, PREFIX_MEGA );
```

Example: Dimensions

Basic Dimensions

```
DLENGTH    : SingleDimension = ZERO_DV ++ { <Length>      |-> 1 };
DMASS      : SingleDimension = ZERO_DV ++ { <Mass>        |-> 1 };
DTIME      : SingleDimension = ZERO_DV ++ { <Time>        |-> 1 };
DCURRENT   : SingleDimension = ZERO_DV ++ { <Current>     |-> 1 };
DTEMP      : SingleDimension = ZERO_DV ++ { <Temperature> |-> 1 };
DAMOUNT    : SingleDimension = ZERO_DV ++ { <Amount>      |-> 1 };
DINTENSITY: SingleDimension = ZERO_DV ++ { <Intensity>   |-> 1 };
```

Pure Quantities

```
DRADIAN     : DimensionlessVector = dim_comp(DLENGTH, dim_inv(DLENGTH)); --L*L**-1
DSTERADIAN  : DimensionlessVector = dim_comp(DAREA, dim_inv(DAREA));     --L**2*L**-2
DWATT       : DimensionVector     = dim_comp(DAREA, dim_comp(DMASS, dim_inv_n(DTIME, 3)));
```

```
DAREA         : DimensionVector = dim_comp_n(DLENGTH, 2);                          --L**2
DVOLUME       : DimensionVector = dim_comp_n(DLENGTH, 3);                          --L**3
DFREQUENCY    : DimensionVector = dim_inv(DTIME);                                  --T**-1
DVELOCITY     : DimensionVector = dim_comp(DLENGTH, DFREQUENCY);                   --L*T**-1
DACCELERATION : DimensionVector  = dim_comp(DVELOCITY, DFREQUENCY);                --L*T**-2
DENERGY       : DimensionVector = dim_comp(DAREA, dim_comp(DMASS, dim_inv_n(DTIME, 2)));  --L**2*M*T**-2
DPOWER        : DimensionVector = dim_comp(DAREA, dim_comp(DMASS, dim_inv_n(DTIME, 3)));  --L**2*M*T**-3
DFORCE        : DimensionVector = dim_comp(DLENGTH, dim_comp(DMASS, dim_inv_n(DTIME, 2))); --L*M*T**-2
DPRESSURE     : DimensionVector = dim_comp(dim_inv(DLENGTH), dim_comp(DMASS, dim_inv_n(DTIME, 2)));  --L**-1*M*T**-2
DCHARGE       : DimensionVector = dim_comp(DINTENSITY, DTIME);                     --I*T
DPDIFFERENCE  : DimensionVector = dim_comp(DAREA,                                  --L**2*M*T**-3*I**-1
                                      dim_comp(DMASS, dim_comp(dim_inv_n(DTIME, 3),
                                      dim_inv(DINTENSITY))));
DCAPACITANCE  : DimensionVector = dim_comp(dim_inv(DAREA),                         --L**-2*M**-1*T**4*I**2
                                      dim_comp(dim_inv(DMASS),
                                      dim_comp(dim_comp_n(DTIME, 4),
                                      dim_comp_n(DINTENSITY, 2))));

DRADIAN      : DimensionlessVector = dim_comp(DLENGTH, dim_inv(DLENGTH)); --L*L**-1
DSTERADIAN   : DimensionlessVector = dim_comp(DAREA, dim_inv(DAREA));     --L**2*L**-2
DWATT        : DimensionVector     = dim_comp(DAREA, dim_comp(DMASS, dim_inv_n(DTIME, 3)));  --L**2*M*T**-3
```

Derived Dimensions

Example: British Imperial System (BIS)

```
values
    BIS_UNIT    : UnitSystem = "BIS";
    --@doc British Imperial System; choose Rankine instead of Farenheit for offset simplicity
    BIS          : ConversionSchema = CONV_ID ++ {
                                                  <Length> |-> 0.9143993,
                                                  <Mass> |-> 0.453592338,
                                                  <Temperature> |-> 5/9
                                              };

types
    --@doc an BIS measurement system has an BIS conversion

    BIS_MeasurementSystem = MeasurementSystem
    inv ms == ms.schema = BIS and ms.unit = BIS_UNIT;


    Yard = BIS_MeasurementSystem
    inv ms == ms.quantity.dim = DLENGTH;


    Pound = BIS_MeasurementSystem
    inv ms == ms.quantity.dim = DMASS;


    Rankine = BIS_MeasurementSystem
    inv ms == ms.quantity.dim = DTEMP;


    BISVolume = BIS_MeasurementSystem
    inv ms == ms.quantity.dim = DVOLUME;


    BISVelocity = BIS_MeasurementSystem
    inv ms == ms.quantity.dim = DVELOCITY;

values
    BIS_YARD     : Yard        = mk_MeasurementSystem(UNIT_LENGTH  , BIS, BIS_UNIT);
    BIS_POUND    : Pound       = mk_MeasurementSystem(UNIT_MASS    , BIS, BIS_UNIT);
    BIS_RANKINE  : Rankine     = mk_MeasurementSystem(UNIT_TEMP    , BIS, BIS_UNIT);
    BIS_VOLUME   : BISVolume   = mk_MeasurementSystem(UNIT_VOLUME  , BIS, BIS_UNIT);
    BIS_VELOCITY: BISVelocity= mk_MeasurementSystem(UNIT_VELOCITY, BIS, BIS_UNIT);
```

Conversion Schema

Measurement System Types

Measurement System

```
imperialise: MeasurementSystem -> Quantity
imperialise(ms) == ms_conv(ms, BIS).quantity;

Launch | Debug
BIS_FOOT: () -> Yard
BIS_FOOT() == scaleMS(1/3, BIS_YARD);

Launch | Debug
BIS_INCH: () -> Yard
BIS_INCH() == scaleMS(1/12, BIS_FOOT());

Launch | Debug
BIS_MILE: () -> Yard
BIS_MILE() == scaleMS(1760, BIS_YARD);

Launch | Debug
BIS_ACRE: () -> BISVolume
BIS_ACRE() == scaleMS(4840, BIS_VOLUME);

Launch | Debug
BIS_OUNCE: () -> Pound
BIS_OUNCE() == scaleMS(1/12, BIS_POUND);
```

General Conversion Function

Scaling Functions

Additional Notes

- Alternate non-decimal systems (British Imperial) or Date/Time, etc.
- Common constants (e.g. speed of light, Planck, Avogadro, etc.)
- Equivalent quantities in different dimensions are demonstrated
- Checking functions for creation of new / corresponding quantities (e.g. pressure per volume = energy; $Pa*m^3$ = Joule = $kg*m^2/s^2$)
- ISQ library works with VDMJ high precision
  - Approximation functions needed for high-precision calculations

Checking functions example

```
> script ./src/main/resources/ISQ.script

p let PA = ms_div(KILOGRAM, SI_ACCELERATION) in
             mk_(is_Pressure(PA), si_dim_view(PA))
           = mk_(          false, "( kg (s**2)  ) / m ")

p let PA = ms_div(KILOGRAM, ms_times(METER, ms_itself_n(SECOND, 2))) in
             mk_(is_Pressure(PA), si_dim_view(PA))
           = mk_(           true, "kg  / ( m (s**2)  )")

p let EPV = ms_times(PA, SI_VOLUME) in
              mk_(is_Energy(EPV), si_dim_view(EPV))
            = mk_(          true, "( (m**2) kg  ) / (s**2) ")
```

Origins and Applications

- Originally motivated by personalised medicine work
  - Prescription given as 8mg of medicine X every 8 hours for 3 weeks
  - Yet BNF (British National Formulary) given as 2.4g of X every 24hrs per month
- High precision smart contract calculations
  - Solidity smart contract DSL for financial instrument conversions
- Potentially useful for FMI FMUs?
  - Conversion between various physical quantities

- Inspired by corresponding Isabelle/HOL implementation by S. Foster.

Future Work

Exploring use into industrial applications

Expanding list of units defined to include

- All units in the SI tables
- Physical quantities list

Further examples within industrial applications

Thanks for Listening