International System of Quantities library in VDM

Leo Freitas[1]

School of Computing, Newcastle University,
leo.freitas@newcastle.ac.uk

Abstract. The International System of Quantities (ISQ) standard was published in 1960 as an effort to tame the wide diversity of measuments systems being developed across the world, such as the centimeter-gram-second versus the meter-kilogram-second for example. Such standard is highly motivated by the potential of "trivial" (rather error-prone) mistakes in converting between incompatible units. There has been such accidents in space missions (NASA's Mars Climite Orbiter), medical devices, etc. Thus, rendering modelling or simulation experiments unusable or unsafe. We address this problem by providing a **SAFE**-ISQ VDM-library that is: **S**imple, **A**ccurate, **Fast**, and **E**ffective. It extends an ecosystem of other VDM mathematical toolkit extensions, including a translation and proof environment for VDM in Isabelle [1].

Keywords: VSCode, VDM, ISQ, Measurement, Libraries

1 Introduction

The International System of Quantities (ISQ) standard[2] (also known as SI[3]) represents the world's most widely used system of measurement. It has official status in nearly every country in the world, and is employed in a wide range of fields (*e.g.* science, technology, medicine, commerce, avionics, everyday use, *etc.*). It is fundamental for both physics and engineering [2]. An interesting compendium of application and their relevance in various domains can be found in [6].

The standard comprises an elegant, minimal and coherent system of units of measurement, with the following seven base unit dimensions of length (in metres, **m**), mass (in kilograms, **kg**), time (in seconds, **s**), electric current (in amperes, **A**), thermodynamic temperature (in kelvins, **K**), amount of substance (in mols, **mol**) and luminous intensity (in candelas, **cd**). From these base units, the system can accomodiate for an unlimited number of aditional (combined and converted) quantities (see Table 1).

Unit combinations are known as coherent derived units (or measurement systems) and can be represented as products of powers of other units. For instance, one can combine length and time to create units of velocity ($\frac{m}{s}$) and acceleration ($\frac{m}{s^2}$); and one can convert between units to create alternative interpretations of units like velocity in

[1] https://github.com/leouk/VDM_Toolkit/
[2] https://en.wikipedia.org/wiki/International_System_of_Quantities
[3] https://en.wikipedia.org/wiki/International_System_of_Units

kilometers or miles per hour. Importantly, these combinations give rise to new units, and convertions give rise to (mostly bijective) unit correspondencess.

Over twenty such coherent derived units are standard, named and routinely used. Further new (unamed) units can still be created (*e.g.* velocity as centimeter per year).This expressivity gives rise to multiple prefixes (*e.g.* kilo, centi, *etc.*) and industry-standard conversion and combination schemes (*e.g.* $\frac{km}{h}$ in $\frac{miles}{h}$).

The Vienna Development Method (VDM) has has been widely used both in industrial contexts and academic ones covering several domains of the fields of Security [4, 5], Fault-Tolerance [8], Medical Devices [7], among others. We extend VDM specification support with a suite of tools and mathematical libraries [4]. The work is also integrated within the VDM Visual Studio Code (VS Code) IDE.

We were inspired to develop the `ISQ.vdmsl` library due to the frustration repetead error-prone (and tediuos) calculations involving unit use and conversions. The `ISQ` VDM library has been used in various industrial applications, such as embedded control systems, medical devices, chemical quantities composition (in drug dosage computation), complex scheduling time-scales, *etc*.

In this paper, we report on the recent extension of the VDM toolkit to support ISQ standard, various conversion schemes, approximate precision, all base and derived representations, *etc.*. We illustrate its use with a variety of scenarios frequently seen in practical applications of ISQ.

2 Background

ISQ has conventions corresponding to measurement-quantity category, abbreviations, dimensions and common names listed in Table 1.

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**

Table 1: Standard measurement system (SI) base quantities of measurement

[4] https://github.com/leouk/VDM_Toolkit/

ISQ unit convertions and combinations are notosrioulsy error-prone, with well known examples of tragic accidents as a result of miscalculations[5]. For critical applications relying on such units conversions and combinations, this is particularly problematic. This motivated the creation of a formal environment to capture ISQ clearly and concisely. For example, multiple conversion schemes are available, such as British Imperial System to the stadard one (SI).

The `ISQ` VDM library is heavily inspired by similar work provided for the Isabelle/HOL theorem prover [3]. We integrate this library within the VDM VSCode extension (https://marketplace.visualstudio.com/items?itemName=overturetool.vdm-vscode), as well as VDMJ [1].

3 Design Principles

Our VDM ISQ library has four core design principles:

1. **S**imple: ease of use is crucial, given ISQ units combination and convertion are pervasive and rather menial, albeit error prone, task;
2. **A**ccurate: ISQ conversion errors account for a considerable amount of modelling process inaccuracies; hence, we wanted a solution where multiple units and correspondences were possible;
3. **F**ast: ISQ conversions can be numerous; hence, a computationally efficient solution is important;
4. **E**ffective: there are multiple ISQ combinations possible; hence, practical use for a variety of such variations is important.

These **SAFE** principles underpin the overall library design goals. Its architecture follows corresponding Isabelle/HOL ISQ structures [3]. We also provide examples of more exotic measurement systems and conversion operators and order of magnitude-dependent approximation functions. Finally, we state some of the expected combination and conversion relationships as traces to be validated. These correspond in part to some of the theorems about ISQ properties as defined in the Isabelle/HOL ISQ library [3].

4 Library architecture

Next, we present how the library architecture. User models have to import `ISQ.vdmsl`, which provides various functionalities divided in six parts:

1. Dimensions (*e.g.* length), dimension vectors (*e.g.* pressure as $\frac{k}{ms^s}$), and operators;
2. Quantities and operators;
3. Measurement systems and operators;
4. Base and derived units;
5. Scaling and conversion over quantities and measurement systems;
6. Common prefixes (*e.g.* alternative names); and
7. Non-decimal measurement systems (*e.g.* British Imperial System).

[5] https://www.simscale.com/blog/nasa-mars-climate-orbiter-metric/

ISQ dimensions and dimension vectors

ISQ dimensions and dimension vectors represent base units of measurment and how they are combined and converted. There are seven base units of measurement as: **L**ength, **M**ass, **T**ime, Electric Current (**I**), Temperature (**Θ**), Amount of Substance (**N**), and Luminous Intensity (**J**). Their standard quantity are, respectively: meter (m), kilogram (kg), second (s), ampere (A), kelvin (K), mole (mol) and candela (cd). Their relationships and how they used for combination and conversions is illutrated in Figure 1.

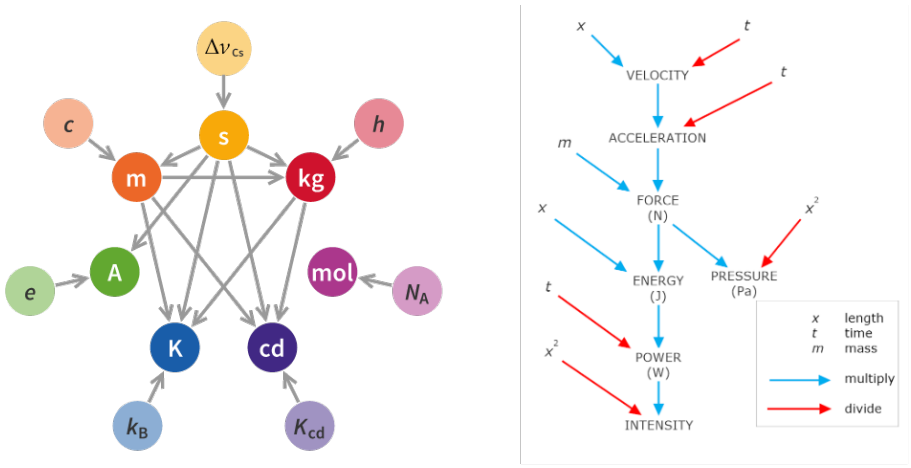

Fig. 1: `ISQ.vdmsl` units of measurement and their conversion relationships.

Dimension vectors are maps capturing how each dimension relates. For instance, the notion of pressure can be measured in a derived unit named *Pascal*, which is kilo per metre per second per second ($\frac{kg}{ms^2}$).

These base unit combinations are captured in a so-called dimension vector, which is modelled as a VDM map, from dimension to an integer that captures the magnitude of each unit. In the *Pascal* case, a single notion of mass (M) and corresponding inverted notions of length ($\frac{1}{L}$) and time ($\frac{1}{T^2}$) as ($\frac{M}{LT^2}$):

```
{ <Length> |-> -1, <Mass> |-> 1, <Time> |-> -2, ... }
```

This dimension mapping enables the representation of a wide variety of base and derived units with many dimensions. To make conversion operations (see below) homogeneous and compositional, dimension vectors must describe all base unit dimensions. In the example above, all remaining base units are mapped to zero.

There are twenty two predefined dimension vectors: seven for each of the base units mentioned; and fifteen for other so-called coherent derived units. These derived units combine one or more of the base units and are for: area (L^2), volume (L^3), frequency ($\frac{1}{T}$), velocity ($\frac{L}{T}$), acceleration ($\frac{L}{T^2}$), energy ($\frac{L^2M}{T^2}$), power ($\frac{L^2M}{T^3}$), force ($\frac{LM}{T^2}$), pressure

($\frac{M}{LT^2}$), charge (IT), potential difference ($\frac{L^2M}{T^3I}$), capacitance ($\frac{T^4I^2}{L^2M}$), radian ($\frac{L}{L}$), steradian ($\frac{L^2}{L^2}$) and wattage ($\frac{L^2M}{T^3}$). Each of these derived units has a corresponding dimension vector as a VDM map constant.

It is also possible to define dimensionless vectors for so-called bare or pure quantities, such as radians. Even though they are non-SI units, such quantities are useful for various applications. These are dimension vectors where all dimensions map to zero. Single quantities like meter or seconds, have dimension vectors with all other dimensions mapped to zero, but the one dimension in question that is mapped to one.

Dimension vector operators are used to combine these derived dimensions, as well as to create new user-defined dimentions. These are recursive map operations, which ensure quantity scaling and conversion will be sound. Moreover, to ensure sound dimension conversions, dimension vectors must be total on all seven base units.

ISQ quantities

ISQ quantities represent the magnitude of a specific dimension vector. Unit quantities (of magnitude 1) for each of the base and derived units are used to compute conversions between measurement systems (*e.g.* kilometer per hour into miles per hour). For example, the unit quantity of area is defined as

```
UNIT_AREA: Quantity = mk_Quantity(1, {<Length> |-> 2, ...});
```

where all other base units in the area dimension vector are mapped to zero.

Quantities comparison (or ordering) are also limited to those within the same dimension vector, given that it does not make sense to compare units of area and velocity, for example:

```
Quantity :: m: Magnitude d:- DimensionVector
ord q1 < q2 == q1.m < q2.m;
```

A variety of other kinds of auxiliary quantities are defined, such as integer quantities, or single dimension quantities for each base and derived units.

Operators between quantities include (but are not limited to): multiplication, replication, division, inversion, summation, (unary and binary) subtraction, scaling, *etc*. They operate over the quantity's magnitude and perform any necessary dimension composition. Finaly, dimension composition operates in the same way as for quantities. For example, if users divide ten meters by two hours, they get the result in meters per hour, which is in fact just meter per second (the base units of length and time).

ISQ measurement systems

An ISQ measurement system represents a named quantity with a corresponding dimension vector conversion schema. A conversion schema defines (non-zero magnitude) factors to convert between other named measurement systems, for all base dimensions.

Like demension vectors, conversion schemas are VDM maps from all dimensions to a non-zero (real-valued) magnitude. They represent a mapping determining how each

dimension magnitude is to be viewed within another measurement system. For example, the standard ISQ conversion schema (`SI`) is simply the identity map of all dimensions (*e.g.* one metter corresponds to one meter). More interestingly, the conversion schema for the British Imperial System (*e.g.* mile, yard, foot, *etc.*) is named `BIS`. The measurement system representing yards in `BIS` is defined as

```
BIS  = { <Length> |-> 0.9143993, <Mass> |-> 0.453592338, <Temp> |-> 5/9, ... };
YARD = mk_MeasurementSystem(mk_Quantity(1, LENGTH), BIS, "BIS");
```

That is, a yard-unit quantity in `BIS` corresponds (will be converted into) 0.9143993 meters in the standard measurement system. The `LENGTH` dimension vector corresponds to dimension vector that maps all dimensions to zero and length to one.

Crucialy, all ommited dimensions in `BIS` map to 1. This is different from dimension vectors that are mapped to zero by default. Such setup is crucial for how quantity dimensions are converted, which will be explained below

With this representation, it is now possible to perform conversions between measurement systems through various operators. There are conversion schema operators for inverse, composion and scaling; and measurement system operators for replication, multiplication, inverse, division and various conversions between different measurement system's schemas.

ISQ base and derived measurement systems

For each of the base and derived dimensions and quantities, we define a corresponding measurement system. They are simple type instantations that represent the named abstractions to be used as types by end users. For example, the standard measurement system for area is defined as

```
Area = MeasurementSystem
inv mk_MesurementSystem(mk_Quantity(magnitude, dimensions), schema, unit) ==
        magnitude = UNIT_AREA.dim and schema = SI and unit = SI_UNIT;
```

where `SI` is the conversion schema that maps all dimensions to 1, and `UNIT_AREA` is the unit quantity for the dimension vector of area described above. Users can then extend this type to define their own notions of area within further refined (type invariants over) `Area` types.

These type extensions are performed for every base and derived dimentions. Furthermore, we define constants for the basic instance of such types (*i.e.* unit version of area within the `SI` standard measurement system). This in turn, enable the measurement system conversions described above, where the `SI` standard measurement system is the baseline.

Further useful measurements are created to illustrate the flexibility of the framework. For instance, the functions below perfom necessary conversions between `BSI` and `SI` measurement systems by scaling.

```
METRE: Metre = mk_MeasurementSystem(UNIT_LENGTH, SI, SI_UNIT);
```

```
SI_YARD: () -> Metre
SI_YARD() == scaleMS(0.9144, METRE);

SI_FOOT: () -> Metre
SI_FOOT() == scaleMS(1/3, SI_YARD());

SI_MILE: () -> Metre
SI_MILE() == scaleMS(1760, SI_YARD());
```

The first constant defines the baseline standard measurement system as a unit of length (*i.e.* a quantity with magnitude 1 and dimension vector mapping length to 1 and all other dimensions to 0). The next functions transforms different imperial measurements by scaling their corresponding measurement system with their numerical magnitude correspondence (*e.g.* foot as third of a yard, yard as 0.9144 of a meter, *etc.*).

This kind of setup serves to illustrate how the various named abstractions can help qualify what computation is needed for what kind of conversion in as an ituitive fashion as possible, hence reducing room for error-prone conversion mistakes.

ISQ scaling and conversion

ISQ scaling is performed by multiplying the magnitude of the entities involved. For `Quantity`, that is direct multiplication of their magnitudes, whereas for measurement systems, it is the scaling of their corresponding quantities, defined as

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

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

That is, measurement system scaling by magnitude m corresponds to scaling the measurement system's quantity magnitude by m.

Of course, when you scale two quantities, there is no change in dimension vector, whereas when you scale a measurement system, you must take dimension conversion into account. If you perform a measurement system operations over two different quantities dimension vectors, then the result is in either a different dimension (*e.g.* if you divide quantities of length and time you get velocity) or in the dimension of one of the parameters (*e.g.* if you sum velocities in different units you must choose one as a resulting dimension vector). For example, if you perform a measurement system summation operation over two different quantities' dimension vectors, the result will be in the dimension of the leading operator (*e.g.* $20\frac{miles}{hour} + 20\frac{km}{hour} =? \frac{miles}{hour}$).

To perform such operations over different measurement systems, a leading measurement system conversion schema has to be considered ($\frac{miles}{hour}$). For the `ISQ.vdmsl` library, that is the conversion schema of the first parameter in a measurement system operator. In the example above, the result would be in miles per hour. To achieve this, we must get the trailing measurement system ($\frac{km}{hour}$) converted into the leading ($\frac{miles}{hour}$) one, then perform quantity scaling.

Given a target onversion schema (`cs_conv`) and a measurement system, we convert the quantity by converting the magnitude according to the target schema (`cs_conv`),

keeping the given schema (`cs`). Next, to quantity-convert in multiple dimensions, we must use the set-product (`prods_r`) of the integer exponenciation between the corresponding conversion schema (`cs(i)`) and dimension vector (`dv(i)`) dimensions, for all base dimensions.

```
quant_conv: ConversionSchema * DimensionVector -> 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_quant_conv: ConversionSchema * MeasurementSystem -> MeasurementSystem
ms_quant_conv(cs_conv, mk_MeasurementSystem(mk_Quantity(m, d), s, u)) ==
    mk_MeasurementSystem(mk_Quantity(quant_conv(cs_conv, d) * m, d), s, u);
```

This is why the defaults for conversion schema and dimension version dimensions are 1 and 0, respectively. Those dimension vector dimensions with zero dimension, will lead to 1 given the exponentiation; others dimensions will be multiplied accordingly to the conversion schema provided conversion mangnitude (`cs(i)`).

The resulting set of magnitudes is then multiplied together, now that their conversion has taken place. The function `quant_conv` is at the heart of how the whole conversion process works: it takes into account the conversion schema for all dimensions, ignoring zero dimensions, with the resulting product being the right magnitude in the right dimension vector.

Dimension vector correspondences. As with other areas of mathematics, the unit signature of operators determine the kind of activity being performed. For example, Joule per Kelvin ($\frac{J}{K}$) is a unit of entropy of heat capacity. Furthermore, the analysis of dimension vectors determines that every coherent (base or derived) SI unit can be written as a unique product of powers of the base units constants.

For example, the *Joule* (J) is defined as the unit of energy as $\frac{kgm^2}{s^2}$. That is, a Joule is equal to the amount of work done when a force displaces a mass (one kg) through a distance (one m) with particular acceleration ($\frac{m}{s^2}$). It can also be viewed as the amount of pressure (in *Pascal*, $Pa = \frac{kg}{ms^2}$) through a volume (m^3). That is, $Pa.m^3$, which is equal to $\frac{kgm^3}{ms^2}$ and can be simplified to a *Joule*.

Finally, an important aspect of unit conversion is the guarantee that the target unit is adequate to the corresponding task intended. That is, a quanity unit type invariant will enforce that the intended computation landed on the intended unit — whichever form its analogous dimensions may take (see Section 5).

ISQ common prefixes

Given that magnitude, quantity and measurement systems are to be used semeinglessly by the user (*i.e.* $10\frac{km}{h}$ as the magnitude 10, quantity 10 of a particular dimension for velocity ($\frac{L}{T}$), or as the quantity within a measurement system of kilometers per hour), these types are known as a `Prefix`.

Prefixes allow users to perform various operations over notions of magnitude, quantity of measurement systems without concern as to whether they are given a magnitude,

a quantity or a measurement system parameter. In practice, this use VDM union types to make the library "natural" to use than having multiple versions of the same computation for slightly different projections of various types involved.

For example, the prefix for `kilo` is defined as the 10^3 magnitude of any given prefix. Then, the scaling of a magnitude for any kind of kilo can be defined as

```
PREFIX_KILO   : MagnitudeN0 = (10**3);

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

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;
```

Now the user can use expressions like `kilo(METRE)` or `kilo(10)` to represent 10^3 units of length in the `SI` measurement system (*i.e.* a kilometer), or 10^4 units of magnitude in whichever quantity or measurement system it gets used. These convenience functions hide from the user the need to perform operations at the level of magnitude, quantity or measurement systems separately.

Specific (canonical `SI`) measurement systems can also be used for standard conversion. For example, the function `metrify` expects a given measurement system quantity, which will be converted to the standard metric quantity (*e.g.* convert miles per hour into meters per second) as

```
metrify: MeasurementSystem -> Quantity
metrify(ms) == ms_conv(ms, SI).quantity;

mph2mps: Magnitude -> Magnitude
mph2mps(mph) == mag(metrify(scaleMS(mph, BIS_MILE_PER_HOUR())));
```

The `BIS_MILE_PER_HOUR` is a measurement system that divide the mile by the hour measurement systems. Overall, even though these computations are somewhat involved, the named abstractions help understanding and introspection of how various concepts are combined and converted.

ISQ other (non-decimal) measurement systems

We define various narrowing types and conversion functions between other (non-decimal) measurement systems. Notably, the one for the British Imperial System (*i.e.* yard, mile, *etc.*) and the one for standard date and time (*i.e.* minute, hour, week, *etc.*).

Users can easily follow the examples to explore other exotic measurement systems they might be interested in representing, where the combination and conversion operators will be straightfoward (and hopefully intuitive) computations.

The library contains 32 defined measurement systems: `SI` 22 basic an derived units; and `BIS` 5 units for length (yards), mass (pound), temperature (rankine), volume ($yard^3$), and velocity ($\frac{yard}{s}$). Also, definitions of the `CGS` measurement system of units for length

(centimeter), and mass (gram); and the `MHC` measurement system of units for mass (miligrams), time (hour), and temperature (celcius). These allows for further derived units like foot, inches, square foot, day, week, year, *etc*.

`ISQ` use and other definitions

We ported the most important theorems from Isabelle/HOL ISQ library [3] and wrote them as VDM traces. We also added various expected equivalences between important measurement systems to hold as part of the library. These are defined through 23 VDM trace specifications, which lead to a total of 2132 trace tests run for nearly all defined measurement systems and their extensiosn. This is useful to ensure computations are working as expected, and also to show users how to use the various library features in practice.

We also define some `SI` common constants, such as the ceaseium frequency, speed of light, plakt constant, *etc*. These also come from the Isabelle/HOL version of this library [3]. Notice that non derived measurement systems are not defined as constants (*e.g.* `SI_WATT`) but as constant functions (*e.g.* `SI_JOULE()`). This is so that they are not computed at initialisation time.

When dealing with (potentially high precision) real-valued operations, it is important to consider approximations. We provide approximation functions for given orders of magnitude, as well as approximate equality module such order of magnitude (*e.g.* `approx_eq(pi, 3.14, 2) = true`). Another useful function we add that is not available in VDM is ceiling of a real value ($\lceil r \rceil$).

5 Applications and Examples

This library was motivated after work various medical applications in a variety of (non standard) units alongside another variety of (familiar yet non-standard) time scales. For example, taking Xmg of drug Y every 8 hours, which then has to be reviewed every 4 weeks for a year. Another concrete application was the use of different time abstractions having to be converted for understanding information within an embedded medical device for organ transplanation.

Dimension vector correspondence checking. The library types enable users to document expected unit correspondences after an operation as type invariants. For example, the energy-pressure-volume (EPV) correspondence discussed above between *Joule* (J, amount of energy) and *Pascal* ($Pa : \frac{kg}{ms^2}$, amount of pressure) one might (mistakenly) write

```
PA = ms_div(KILOGRAM, SI_ACCELERATION);
p is_Pressure(PA)
= false

PA' = ms_div(KILOGRAM, ms_times(METER, ms_itself_n(SECOND, 2)));
p is_Pressure(PA')
= true

EPV = ms_times(PA', SI_VOLUME);
```

```
p is_Energy(EPV)
= true
```

That is, considering mass over acceleration ($\frac{m}{s^2}$) instead of mass dispersion over time. This also illustrate the use of measurement systems multiplication, replication and division. Finally, we check that the *Pascal* measurement system alongside some volume equates to the expected unit of energy. These invariant checks (failures and successes) are useful to ensure that operations are being performed in the expected/adequate measurement system.

Dimension vector inspection. Given such computations over unit correspondences can test one's memory of physics (or other topics), the library also provides some useful utility functions to quickly inspect the dimension vector as one would use in "pen-and-paper" exercises. For instance, the dimension vector for the variables above can be viewed as

```
p si_dim_view(PA)
= "kg * (s**2) / m"

p si_dim_view(PA')
= "kg / m * (s**2)"

p si_dim_view(EPV)
= "kg * (m**2) / (s**2)"
```

This can be useful for debugging whether the unit of the measurement system chosen is the one expected. Notice that `si_dim_view` accepts a `Prefix`, hence making it seemless whether you give it a quantity or measurement system.

Finally, the general `dim_view` function expects an input that maps dimensions to their visualisation strings. For example, we add such a map to change the visualisation into `BIS`. This is not a check the dimension is "correct" with respect to the measurement system conversion schema, but simply a debugging aid.

New measurement systems. We created a new measurement system and conversion schema to cater for a "miligram-hour-celcius" measurement system (`MHC`). It is justs like the `SI` measurement system, yet views mass, time and temperature with these other corresponding units.

```
MHC: ConversionSchema = CONV_ID ++
    { <Mass> |-> mag(milli(milli(KILOGRAM))), <Temperature> |-> -272.15,
      <Time> |-> mag(hour(SI, SI_UNIT)) };

SECOND: Second = mk_MeasurementSystem(UNIT_TIME, SI, SI_UNIT);

second: ConversionSchema * UnitSystem -> Second
second(cs, u) == mk_MeasurementSystem(UNIT_TIME, cs, u);

hour: ConversionSchema * UnitSystem -> Second
hour(cs, u) == scaleMS(60, scaleMS(60, second(cs, u)));
```

The auxiliary functions create a conversion-schema parameterised notion of base unit of time (*e.g.* `SI`'s `SECOND` vs. `second` possibly with different schema). In fact, even

though our desired time baseline was hour, we added other granularities like minute, day, week, year, *etc.* When evaluated the `MHC` system has $\frac{1}{10^6}$ value for mass, 3600 for time, and -272.15 for temperature. That is, usual mass (kilogram) is viewed in miligrams (mg), usual time (in seconds) is viewed in hours, and usual temperature (in kelvin) is viewed in centigrade.

Next, we created application-level types for each dimension within the measurement system, alongside corresponding single-unit constants for each type.

```
Milligram = MHC_MeasurementSystem inv ms == ms.quantity.dim = MASS;
Hour      = MHC_MeasurementSystem inv ms == ms.quantity.dim = TIME;
Celcius   = MHC_MeasurementSystem inv ms == ms.quantity.dim = TEMP;

MGRAM   : Milligram = mk_MeasurementSystem(UNIT_MASS, MHC, MHC_UNIT);
MHOUR   : Hour      = mk_MeasurementSystem(UNIT_TIME, MHC, MHC_UNIT);
MCELCIUS: Celcius   = mk_MeasurementSystem(UNIT_TEMP, MHC, MHC_UNIT);
```

These types and baseline constants were then refined in the actual application(s) to specific constraints like maximum treatment time or temperature. Then, we defined constant functions to convert other used time granularities back to the baseline in hour[6].

```
hDAY: () -> Hour
hDAY()  == scaleMS(HOURS_PER_DAY, MHOUR);
hWEEK() == scaleMS(DAYS_PER_WEEK, hDAY());
hYEAR() == scaleMS(DAYS_PER_YEAR, hDAY());

hMONTH: WhichMonth -> Hour
hMONTH(m) == scaleMS(DAYS_PER_MONTH(m), hDAY());
```

For a view of specific months in hours, we define a map from month to number of days and scale the result with `hDAY()`, hence creating month-specific hour periods. For the application-level we could have written something like (to hide away any ISQ details):

```
n_times_day2every_x_hours: nat1 -> real
n_times_day2every_x_hours(times_a_day) ==
    mag(scaleMS(1/mag(scaleMS(times_a_day, MHOUR)), hDAY()));
```

This function converts times per day in number of hours. Of course, this simple concrete example is somewhat overkill. It serves to show how the library can be used.

For the real application, however, this was crucial, given variations were defined per week, where intake was not always per hour (*e.g.* three times a day), and other more complicated pharmacokenetic interactions existed.

Other examples of use are provided in the `ISQ.vdmsl` distribution [7]. They show various ISQ measurement systems in practical use.

[6] We omitted the same type signature of other functions.
[7] https://github.com/leouk/VDM_Toolkit/

6 Results and discussion

ISQ representation and operations are an important part of modelling tasks within computing, physics and engineering. Unit conversion, combination and manipulation are notoriously error prone, having led to a number of serious accidents[8].

The `ISQ.vdmsl` library closely follows the structure of a similar library for the Isabelle/HOL theorem prover [3]. It has been used in a few practical industrial applications involving emedded systems and medical devices and processes.

In this paper, we presented a formally defined ISQ library in VDM that adheres to our **SAFE** design principles (in Section 3). The library architecture (Section 4) is **S**imple, given its layered access to functionality. It is also **A**ccurate, given the presence of multiple kinds of user-defined invariant and other structural and data validation checks, such as enforcement of specific measurement system characteristics for well known quantities and units. It is **F**ast, as the computations are simple and directly available in VDM itslef. Finally, it is **E**ffective, given its combination of speed, ease of use, multiple capabilities around ISQ measurement systems creation, combination, conversion, scaling, and so on.

Future work. We are interested into applying the library concepts to other industrial applications, such as Functional Mock-up Interfaces, which do have such notions for ISQ units[9]. Moreover, we also plan to extend the physical units to include all those in the SI tables[10] and other more exotic list of units[11].

Acknowledgements. We acknowledge funding from NCSC on verification on payment systems. We very much appreciate fruitiful discussion with and suggestions by Nick Battle about the usefulness of such library and on certain VDMJ needs.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Bureau International des Poids et Mesures, Joint Committee for Guides in Metrology: Basic and general concepts and associated terms (vim) (3rd ed.). Tech. rep., BIPM, JCGM (2012), version 2008 with minor corrections
3. Foster, S., Wolff, B.: A sound type system for physical quantities, units, and measurements. Archive of Formal Proofs (October 2020), https://isa-afp.org/entries/Physical_Quantities.html, Formal proof development
4. Kulik, T., Macedo, H.D., Talasila, P., Larsen, P.G.: Modelling the HUBCAP Sandbox Architecture In VDM – a Study In Security. In: Fitzgerald, J.S., Oda, T., Macedo, H.D. (eds.) Proceedings of the 18th International Overture Workshop. pp. 20–34. Overture (December 2020)

[8] https://www.simscale.com/blog/nasa-mars-climate-orbiter-metric/
[9] https://fmi-standard.org/docs/3.0/#_physical_units
[10] https://en.wikipedia.org/wiki/International_System_of_Units
[11] https://en.wikipedia.org/wiki/List_of_physical_quantities

5. Kulik, T., Talasila, P., Greco, P., Veneziano, G., Marguglio, A., Sutton, L.F., Larsen, P.G., Macedo, H.D.: Extending the formal security analysis of the hubcap sandbox. In: Macedo, H.D., Thule, C., Pierce, K. (eds.) Proceedings of the 19th International Overture Workshop. Overture (10 2021)
6. Laboratory, N.P.: A to Z of Measurement. NPL (2019)
7. Macedo, H.D., Larsen, P.G., Fitzgerald, J.: Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods, 15th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 5014, pp. 181–197. Springer-Verlag (2008)
8. Nilsson, R., Lausdahl, K., Macedo, H.D., Larsen, P.G.: Transforming an industrial case study from VDM++ to VDM-SL. In: Pierce, K., Verhoef, M. (eds.) The 16th Overture Workshop. pp. 107–122. Newcastle University, School of Computing, Oxford (July 2018), TR-1524