



# The Measurement Library: Representing Physical Types in PVS \*

Ashlie B. Hocking and M. Anthony Aiello

Dependable Computing, Charlottesville, VA  
ben.hocking@dependablecomputing.com  
tony.aiello@dependablecomputing.com

## Abstract

Cyber-physical systems model physical phenomena, implicitly or explicitly, in order to interact with the real world. Representation of physical phenomena, including dimensionality and units, using the PVS type system provides users with the ability to create specifications that more accurately describe cyber-physical systems. This paper discusses two related libraries that each present a different approach to providing functionality for using units in PVS. Objectives in creating the libraries include: soundness, ease of use, ease of readability, and effect on provability.

## 1 Introduction

Creating a strong specification of cyber-physical systems is difficult. Correctly describing a cyber-physical system with a formal specification requires an accurate and consistent representation of the physical phenomena shared between the system and the real world. Furthermore, many proofs of properties guaranteed by the specification rely on accurate modeling of these phenomena. Failing to accurately represent these phenomena invalidates proofs that would otherwise be sound. These invalid proofs may lead to an inability to detect failures, which may in turn lead to accidents.

In this paper, we present two approaches to encoding **measurements** in PVS specifications that enable first-class representation of physical phenomena using physical types. When we use the term **measurement**, we refer to a *value* and a **unit**, where units including *dimensionality* (e.g., distance vs. time), *system of measurement* (e.g., metric or imperial), and *scaling* (e.g., km vs. mm). Both approaches are captured in an open-source library, which can be found at the [dcpvslib](#) project on GitHub [5].

Our objectives in creating the library include:

1. Soundness: the library must enable users to detect mistakes, e.g., in unit composition.
2. Readability: the library must enable readers to easily spot mistakes without using the prover.

---

\*Approved for Public Release; Distribution Unlimited (Case Number: 88ABW-2017-1472)

3. Ease of use: the library must make adding measurements easy to use, since ease of use increases likelihood that the library will be applied.
4. Effect on provability: use of the library must not unduly increase the complexity of completing proofs.

Designing approaches that represent systems of units while preventing implicit combination of units from differing systems requires tradeoffs between simplicity and rigor. To explore these tradeoffs, we have created two different approaches, a *system-templated library* (Section 4) and a *system-field library* (Section 5).

## 2 Motivating Examples

To support readability and ease of use, we want to be able to write PVS statements such as those shown in Listing 1.

```
distance: length = 3 * m;
N: force = kg * m / s^2;
piston_pressure: pressure = 3 * N / cm^2;
a_dist: length;
b_dist: length;
c_dist: length = sqrt(a_dist^2 + b_dist^2);
```

Listing 1: Motivating PVS snippets

This notation is not only immediately familiar to readers, but also makes it is easy to combine units. The expressiveness of the PVS type system allows this representation while ensuring that operations on measurements will be type correct. For example, PVS generates type-correctness conditions (TCCs) to verify that dividing Newtons by squared centimeters results in a pressure. Integrating dimensional analysis into type checking is a key feature of this approach: no additional effort is required on the part of users to ensure that measurements are handled correctly in their specifications.

Using measurements, we want to prevent certain types of operations that are indicative of errors. For example, adding meters to centimeters *implicitly* or combining units from different systems of measurement *implicitly* indicates that the specification may be incorrect. Either operation may be correct, but we require that such operations include *explicit* steps so that we can detect errors.

For example, consider a specification (Figure 1) that adds 10 (meters) to 25 (centimeters). Without explicitly defined units, this yields 35 (something) instead of 10.25 (meters) or 1,025 centimeters. While the addition of these two values makes physical sense, in order to avoid ambiguity a specification should define how the result should be represented (meters, centimeters, or something else). An implicit conversion to the specified unit (meters to centimeters or centimeters to meters) could then be assumed by a type-checking library. However, our experience with creating specifications for Simulink models convinces us that requiring the conversion to be explicit (e.g., by multiplying 25 centimeters by 1 meter per 100 centimeters) will reduce the likelihood that mistakes will be made involving units.

Similarly, consider a model of a tank where the height of the liquid in the tank is based on the tank size and the amount of liquid entering and leaving the tank, as shown in Figure 2. The PVS specification for this model is shown in Listing 2.

This PVS theory generates no TCCs; it is trivially type consistent. However, if we add units to this model, as shown in Listing 3 the model generates unprovable TCCs.

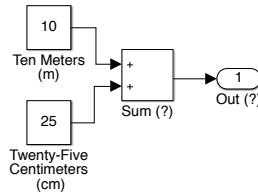


Figure 1: Attempting to add 10 m and 25 cm in Simulink

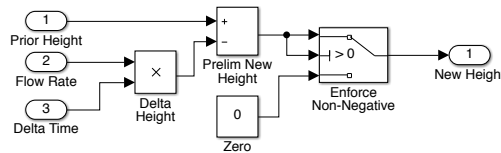


Figure 2: Calculating the new fluid height in a tank

The problem with this model is that multiplying the flow rate (volume per second) times the time (seconds) yields a volume, not a length. For a tank with a cross-sectional area of  $1 \text{ m}^2$ , the new height *can* be calculated by adding to the previous height the volume of liquid entering and the subtracting the volume of liquid leaving. However, the operations in this model do not yield the correct units. Moreover, if these operations were used without review for a tank whose cross-sectional area is not  $1 \text{ m}^2$ , the operations would produce entirely incorrect results.

The correct PVS specification for this model, with units, is provided in Listing 4.

Informal representations of units or dimensionality allow the user to detect this kind of mistake, but require manual verification of the correctness and consistency of measurements. These informal representations are subject to errors of oversight. For example, consider the mistake of dividing by a conversion factor when one should be multiplying by a conversion factor. A careful reader might find this mistake, but a reader who is expecting the formula to be correct could easily overlook the mistake. Formalization of representations of units and dimensionality allows these kinds of mistakes to be detected automatically, increasing assurance that the specification is correct.

### 3 Methods

Specification of a measurement type should include value, dimensionality, scaling, and system of measurement. Measurement types should support basic operations (PMDAS — power, multi-

```
CalcNewHeight(prior_height: real, flow_rate: real,
              delta_time: real): real =
  LET delta_height: real = flow_rate * delta_time IN
  LET prelim_new_height: real = prior_height - delta_height IN
  IF prelim_new_height > 0 THEN
    prelim_new_height
  ELSE
    0
  ENDF;
```

Listing 2: PVS model of tank calculation without units

```

CalcNewHeight(prior_height: length_m,
              flow_rate: volume_per_time_m3_per_s,
              delta_time: time_s): length_m =
  LET delta_height: length_m = flow_rate * delta_time IN
  LET prelim_new_height: length_m =
    prior_height - delta_height IN
  IF prelim_new_height > 0 * m THEN
    prelim_new_height
  ELSE
    0 * m
  ENDIF;

```

Listing 3: PVS model of tank calculation with inconsistent units

```

CalcNewHeight(prior_height: length_m,
              flow_rate: volume_per_time_m3_per_s,
              delta_time: time_s): length_m =
  LET delta_volume: volume_m3 = flow_rate * delta_time IN
  LET delta_height: length_m =
    delta_volume / C_Cross_Sectional_Area IN
  LET prelim_new_height: length_m =
    prior_height - delta_height IN
  IF prelim_new_height > 0 * m THEN
    prelim_new_height
  ELSE
    0 * m
  ENDIF;

```

Listing 4: PVS model of tank calculation with consistent units

plication, division, addition, and subtraction) while preventing invalid operations (e.g., addition of meters and centimeters).

### 3.1 Dimensionality

Dimensionality includes those aspects identified by the International System of Units:

- Length
- Mass
- Time
- Electric Current
- Thermodynamic Temperature
- Amount of Substance
- Luminous Intensity
- Angle

The measurement libraries are designed to be extensible: additional dimensionality aspects can be added as required. Similar to work done elsewhere (see Subsection 6.2), dimensionality can be thought of as a vector of powers so that length (e.g., 1 m) corresponds to  $[1, 0, 0, 0, 0, 0, 0]$  and speed (e.g., 1 m/s) to  $[1, 0, -1, 0, 0, 0, 0]$ . This representation makes operations on measurements straightforward.

## 3.2 Scaling

Scaling is a conversion factor from an arbitrary scale factor to a base representation. For example, if meters is the base representation, then centimeters will have a scale factor of 0.01, so that 10 centimeters = 10 x 0.01 meters. As another example, since watts are defined as  $\text{kg} \cdot \text{m} / \text{s}^2$ , if grams, meters, and seconds are used as a base representation, then watts will have a scale factor of 1,000 and kilowatts will have a scale factor of 1,000,000. Scaling provides two benefits:

1. a means of discriminating meters (scale = 1) from centimeters (scale = 0.01), and
2. a means of supporting custom measurements, such as *RPM per 5 ms* [6].

## 3.3 Operations

### 3.3.1 Multiplication/Division

Multiplying (dividing) measurement  $m_1$  by  $m_2$  should return a measurement with the following properties:

- a value equal to the value of  $m_1$  multiplied (divided) by the value of  $m_2$ ,
- dimensions equal to the dimensions of  $m_2$  added to (subtracted from) the dimensions of  $m_1$ ,
- the same system of measurement as  $m_1$  and  $m_2$ , and
- a scale equal to the scale of  $m_1$  multiplied (divided) by the scale of  $m_2$ .

A precondition of multiplying (dividing)  $m_1$  by  $m_2$  is that they belong to the same system of measurement.

### 3.3.2 Exponentiation

The operation of raising a measurement  $m$  to a power  $p$  should return a measurement with the following properties:

- a value equal to the value of  $m$  raised to the power  $p$ ,
- dimensions equal to the dimensions of  $m$  multiplied by the power  $p$ ,
- the same system of measurement as  $m$ , and
- a scale equal to the scale of  $m$  raised to the power  $p$ .

Negative exponentiation should obviously be allowed, for example to support speed (distance per time). However, less obviously, non-integer exponentiation should also be supported, for example to support the statcoulomb, which is represented in CGS as  $\text{cm}^{3/2} \text{g}^{1/2} \text{s}^{-1}$ .

### 3.3.3 Addition/Subtraction

Adding (subtracting) measurement  $m_1$  and  $m_2$  should return a measurement with the following properties:

- a value equal to the value of  $m_2$  added to (subtracted from) the value of  $m_1$ ,
- dimensions equal to the dimensions of  $m_1$  and  $m_2$ ,
- the same system of measurement as  $m_1$  and  $m_2$ , and
- the same scale as  $m_1$  and  $m_2$ .

Preconditions of adding (subtracting)  $m_1$  and  $m_2$  are:

- $m_1$  and  $m_2$  must have the same dimensions,
- $m_1$  and  $m_2$  must have the same system of measurement, and
- $m_1$  and  $m_2$  must have the same scale.

### 3.3.4 Comparisons

Comparing measurement  $m_1$  and  $m_2$  naturally requires that the two measurements have the same dimensions. As discussed in Section 2, our goal is to *not* allow implicit conversions between units with different scales (e.g., m and cm). To that end, comparisons also require that the two measurements have the same scale and system of measurement. While we could allow comparison between measurements with different scales by having the comparison be between the *base values*, or the product of the scale and the value, such comparisons would not yield the same results as the comparisons done within typical model-based development packages or in typical programming languages (unless explicitly including a package that supports units). E.g., if a Simulink model compared a constant of 2 m and a constant of 15 cm, it would find that  $15 \text{ cm} < 2 \text{ m}$ , so this is not a comparison we support. Thus, comparisons have the same preconditions as addition and subtraction. Note that such comparisons can still be made by performing explicit conversions, so that one could verify that  $15 * \text{cm} * \text{m} / (100 * \text{cm}) < 2 * \text{m}$ . Similarly, if one wants to check to see if  $10 * \text{m} = 1000 * \text{cm}$ , one first needs to explicitly convert the equality to  $10 * \text{m} * 100 * \text{cm} / \text{m} = 1000 \text{ cm}$  in order to properly form the question. As with addition, our goal is to reduce the chances of inadvertently using the wrong units, and in our experience requiring explicit conversions between units with different scales (or systems of measurement) is the best way to achieve this.

## 3.4 System of Measurement

Systems of measurement represent a set of base units, standard combinations of those units, and rules relating them to each other. The measurement library current supports the SI and Imperial systems of measurement. Other systems can be added by users.

In this paper, we discuss two approaches to specifying a system of measurement:

1. specify the system of measurement as a template variable to the theory, and
2. specify the system of measurement as a field in the measurement type.

We discuss the advantages and disadvantages of these two approaches in the following sections.

## 4 System-Templated Library

The system-templated library is parameterized by `measurement_systems`: an enumeration of `METRIC`, `IMPERIAL`, as shown in Listing 5.

```
measurements[
  (IMPORTING measurement_systems) S: system_enum]: THEORY
```

Listing 5: Header of the measurement theory in the system-templated library

Within this library a measurement type is defined as shown in Listing 6.

```
measurement: TYPE =
  [#
    value:      real,
    scaling:    posreal,
    length_dim: real,
    mass_dim:   real,
    time_dim:   real,
    current_dim: real,
    temp_dim:   real,
    intens_dim: real,
    angle_dim:  real
  #];
```

Listing 6: Measurement type in the system-templated library

Units are defined in this library by instantiated versions of templated theories, where different systems use different scaling factors. For example, the `imperial_lengths` theory is defined in Listing 7.

```
imperial_lengths: THEORY
BEGIN

  IMPORTING measurement_systems;
  IMPORTING lengths [IMPERIAL];

  ft_to_m: real = 0.3048;

  ft: poslength = unit_length WITH ['scaling := ft_to_m];
  inch: poslength = ft WITH ['scaling := ft_to_m * 1/12];
  yard: poslength = ft WITH ['scaling := ft_to_m * 3];
  mi: poslength = ft WITH ['scaling := ft_to_m * 5280];

END imperial_lengths
```

Listing 7: The imperial lengths theory

### 4.1 Predicates

In this approach, the predicates `dimension_match?` and `unit_match?` are required to support operations on measurements. The `dimension_match?` predicate is shown in Listing 8, and tells us that two measurements have identical dimensions.

The `unit_match?` predicate, which uses the `dimension_match?` is shown in Listing 9 and is true when two measurements have the same dimensions and scale.

We also define a `comparable?` predicate that is identical to the `unit_match?` predicate to support the comparison operations discussed in Subsection 3.3.4

In addition to the generic predicates defined above, measurement subtypes (e.g., `length`) have predicates defined as part of their type definition. For example, the type `length` is defined in terms of the predicate `length?` shown in Listing 10.

```

dimension_match?(x: measurement, y: measurement): bool =
  (x`length_dim = y`length_dim) AND
  (x`time_dim = y`time_dim) AND
  (x`mass_dim = y`mass_dim) AND
  (x`current_dim = y`current_dim) AND
  (x`temp_dim = y`temp_dim) AND
  (x`intens_dim = y`intens_dim) AND
  (x`angle_dim = y`angle_dim);

```

Listing 8: Specification of `dimension_match?` predicate

```

unit_match?(x: measurement, y: measurement): bool =
  dimension_match?(x, y) AND
  (x`scaling = y`scaling);

```

Listing 9: Specification of `unit_match?` predicate

## 4.2 Operations

### 4.2.1 Multiplication/Division

Multiplication and division are defined by the rules provided in Subsection 3.3.1. The specification of the multiplication operation is shown in Listing 11. Because the system of measurements is specified as a theory template variable, the precondition that the systems match is not explicit.

### 4.2.2 Exponentiation

Exponentiation is defined in terms of multiplication (or division for negative powers). The specification of the  $\wedge$  operation is shown in Listing 12. While the function `expt` only allows non-negative integers, the  $\wedge$  operator handles all integers.

Fractional powers can be attained through the use of `sqrt`, as shown in Listing 13. While using integer exponentiation and square roots to create units such as the statcoulomb discussed in Subsection 3.3.2 is somewhat awkward, these units can be defined once and reused with ease. Currently, the library does not support arbitrary powers (e.g., one-third), but this is only because there has not yet been a reason to support arbitrary powers.

### 4.2.3 Addition/Subtraction

Addition and subtraction are defined by the rules provided in Subsection 3.3.3. The specification of the addition operator is shown in Listing 14. The `unit_match?` predicate is used to guarantee the precondition requirements for addition that the two measurements have the same dimensions and scale.

### 4.2.4 Comparisons

Comparison operations (e.g., `<`) have the same preconditions as addition/subtraction. For example, the `<` operator is defined as shown in Listing 15, where `base_value` is the product of `value` and `scaling`.

```

% or distance, width, etc.
length?(m: measurement): bool =
  dimension_match?(m, zero_measurement
    WITH [ `length_dim := 1]);

```

Listing 10: Specification of `length?` predicate



```

*(x: measurement, y: measurement): measurement =
  (#
    value      := x`value * y`value,
    scaling    := x`scaling * y`scaling,
    length_dim := x`length_dim + y`length_dim,
    time_dim   := x`time_dim + y`time_dim,
    mass_dim   := x`mass_dim + y`mass_dim,
    current_dim := x`current_dim + y`current_dim,
    temp_dim   := x`temp_dim + y`temp_dim,
    intens_dim := x`intens_dim + y`intens_dim,
    angle_dim  := x`angle_dim + y`angle_dim
  #);

```

Listing 11: Specification of the multiplication operator

```

% Scaling is exponentiated via multiplication rule
expt(x: measurement, n: nat): RECURSIVE measurement =
  IF n = 0 THEN
    unit_measurement
  ELSE
    x * expt(x, n - 1)
  ENDIF
MEASURE n;

zero_value?(m: measurement): bool = (m`value = 0);

^(x: measurement,
  i: {i: int | NOT(zero_value?(x)) OR i >= 0}):
  measurement =
  IF i >= 0 THEN expt(x, i) ELSE 1 / expt(x, -i) ENDIF;

```

Listing 12: Specification of the  $\wedge$  operator

### 4.3 Analysis

Unfortunately, with the system-templated library, it becomes possible to inadvertently combine units from different systems so that `m * ft` is valid. One *could* define a predicate to check for whether a unit is consistently scaled as a power of ten as shown in Listing 16.

Because imperial units are always defined with a scaling factor that is not a power of ten (with the exception of units that are system-agnostic such as seconds), measurements that are metric can be identified by the `power_of_ten_measurement?` predicate. However, this predicate will miss certain situations where custom metric units have scaling factors that are *not* powers of ten. The decision to use `METRIC` as a baseline (so that `m` has a scaling factor of 1 and `ft` has a scaling factor of 0.3048, for example) instead of `IMPERIAL` was made primarily for our preference of the metric system, but is also supported by the potential utility of the `power_of_ten_measurement?` predicate.

The system-field library addresses this limitation.

```

sqrt(x: nnmeasurement): nnmeasurement =
  (#
    value      := sqrt(x`value),
    scaling    := sqrt(x`scaling),
    length_dim := x`length_dim / 2,
    time_dim   := x`time_dim / 2,
    mass_dim   := x`mass_dim / 2,
    current_dim := x`current_dim / 2,
    temp_dim   := x`temp_dim / 2,
    intens_dim := x`intens_dim / 2,
    angle_dim  := x`angle_dim / 2
  #);

```

Listing 13: Specification of the `sqrt` operation

```
+ (x: measurement,
  y: {m: measurement | unit_match?(x, m)}):
  {m: measurement | unit_match?(x, m)} =
  x WITH [ `value := x`value + y`value ]
```

Listing 14: Specification of the addition operator

```
< (x: measurement, y: {m: measurement | comparable?(x, m)}): bool =
  base_value(x) < base_value(y);
```

Listing 15: Specification of the less than operator

## 5 System-Field Library

The system-field library uses a field to track the system of measurement. In the system-field library, a measurement type is defined as shown in Listing 17.

In this library, possible values for `system_enum` are `NOT_APPLICABLE` (for dimensionless measurements), `ANY` (for units that are system agnostic, such as seconds), `METRIC`, and `IMPERIAL`. Units are defined in this library by specifying the appropriate system of measurement (and scaling where applicable) as shown in Listing 18.

Both meters and other lengths are defined in the `lengths` theory, with other lengths shown in Listing 19.

As mentioned previously, some units are defined to be system agnostic, such as `s` defined in the `times` theory and `Hz` defined in the `frequencies` theory.

### 5.1 Predicates

The system-field library requires far more predicates. While the `dimension_match?` predicate in this library is defined identically to the `dimension_match?` predicate in the system-templated library, the `unit_match?` predicate requires multiple helper predicates. The `explicit_system?` predicate is used to separate the `METRIC` and `IMPERIAL` explicit systems of measurement from the `NOT_APPLICABLE` and `ANY` `system_enum` options. Because the `NOT_APPLICABLE` option is only valid for dimensionless quantities, not all measurements are valid. The `valid_measurement?` predicate, shown in Listing 20 is true for valid measurements.

In the system-field library, determining whether two systems “match” is more complicated than determining if the systems are identical. The `system_match?` predicate shown in Listing 21 is true if two measurements have matching systems.

Note that the `system_match?` predicate is not transitive. For example, if measurement  $x$  is `METRIC`, measurement  $y$  is `ANY`, and measurement  $z$  is `IMPERIAL`, then `system_match?(x, y)` is true, and `system_match?(y, z)` is true, but `system_match?(x, z)` is not.

The `unit_match?` predicate is shown in Listing 22.

The `preferred_system?` predicate is used when determining what system of measurement to use for the result of mathematical operations. For example, if dividing two measurements and the first measurement is `METRIC` and the second measurement is `ANY`, then the first measurement is the preferred system and the `preferred_system?` predicate will be true.

```
% i can be < 0
power_of_ten_measurement?(m: measurement): bool =
  EXISTS(i: int): (10^i = m`scaling);
```

Listing 16: Specification of the `power_of_ten_measurement` predicate

```

measurement: TYPE =
[#
  value:      real,
  system:    system_enum,
  scaling:    posreal,
  length_dim: real,
  mass_dim:   real,
  time_dim:  real,
  current_dim: real,
  temp_dim:  real,
  intens_dim: real,
  angle_dim: real
#];

```

Listing 17: Measurement type in the system-field library

```
m: poslength = unit_length WITH [`system := METRIC];
```

Listing 18: Specification of meters

```

cm: poslength = m WITH [`scaling := 1/100];
mm: poslength = m WITH [`scaling := 1/1000];

% Imperial
ft: poslength = unit_length WITH [`system := IMPERIAL];
inch: poslength = ft WITH [`scaling := 1/12];
mi: poslength = ft WITH [`scaling := 5280];

```

Listing 19: Specification of various length units

```

valid_measurement?(m: measurement): bool =
(m`system /= NOT_APPLICABLE) OR
(dimension_match?(zero_measurement, m))

```

Listing 20: Specification of the `valid_measurement?` predicate

```

system_match?(x: measurement, y: measurement): bool =
valid_measurement?(x) AND valid_measurement?(y) AND
((x`system = y`system) OR (NOT explicit_system?(x)
OR (NOT explicit_system?(y))));

```

Listing 21: Specification of the `system_match?` predicate

```

unit_match?(x: measurement, y: measurement): bool =
dimension_match?(x, y) AND
system_match?(x, y) AND
(x`scaling = y`scaling);

```

Listing 22: Specification of `unit_match?` predicate

```

preferred_system?(x: measurement, y: measurement): bool =
IF explicit_system?(x) THEN
  TRUE
ELSIFF explicit_system?(y) THEN
  FALSE
ELSIFF (x`system = ANY) OR (y`system = NOT_APPLICABLE) THEN
  % either they're both all, both n/a, or x is all and y is n/a
  TRUE
ELSE
  % x is n/a and y is all
  FALSE
ENDIF

```

Listing 23: Specification of `preferred_system?` predicate

We also define a `comparable?` predicate that is similar to the `unit_match?` predicate, but allows zero values to be compared regardless of whether units match.

In addition to the generic predicates defined above, measurement subtypes (e.g., `length`) have predicates defined as part of their type definition.

## 5.2 Operations

### 5.2.1 Multiplication/Division

Multiplication and division are defined similarly to how they are defined in Subsection 4.2.1. However, because the system of measurements is specified by a field, the precondition that the systems match is explicit. The specification of the multiplication operation is shown in Listing 24. As with addition, the `preferred_system?` predicate is required to ensure that multiplication is commutative.

```
*(x: valid_measurement,
  y: {m: valid_measurement | system_match?(x, m)}):
  {m: valid_measurement | system_match?(x, m)} =
  (#
    value      := x`value * y`value,
    system     := IF preferred_system?(x, y) THEN
                  x`system
                ELSE
                  y`system
                ENDF,
    scaling    := x`scaling * y`scaling,
    length_dim := x`length_dim + y`length_dim,
    time_dim   := x`time_dim + y`time_dim,
    mass_dim   := x`mass_dim + y`mass_dim,
    current_dim := x`current_dim + y`current_dim,
    temp_dim   := x`temp_dim + y`temp_dim,
    intens_dim := x`intens_dim + y`intens_dim,
    angle_dim  := x`angle_dim + y`angle_dim
  #);
```

Listing 24: Specification of the multiplication operator

### 5.2.2 Exponentiation

As shown in Listing 25, exponentiation is defined similarly to how it is defined in Subsection 4.2.2. The primary difference in this library is that the raising a measurement to the zero power results in a dimensionless quantity with scale and value 1, but with the same system of measurement as the original measurement. The choice to preserve the system of measurement, rather than using `NOT_APPLICABLE` or `ANY` was made for both simplicity of implementation and ease of use.

### 5.2.3 Addition/Subtraction

Addition and subtraction are defined similarly to how they are defined in Subsection 4.2.3. The specification of the addition operator is shown in Listing 26. The `preferred_system?` predicate is used to determine which system of units the result should be in cases where one of the measurements is zero-valued and without an explicit system. This predicate is required to ensure that addition is commutative.

```

% Scaling is exponentiated via multiplication rule
expt(x: valid_measurement, n: nat):
  RECURSIVE {m: valid_measurement | x`system = m`system} =
  IF n = 0 THEN
    unit_measurement WITH [ `system := x`system]
  ELSE
    x * expt(x, n - 1)
  ENDIF
  MEASURE n;

zero_value?(m: valid_measurement): bool = (m`value = 0);

^(x: valid_measurement,
  i: {i: int | NOT(zero_value?(x)) OR i >= 0}):
  {m: valid_measurement | system_match?(x,m)} =
  IF i >= 0 THEN expt(x, i) ELSE 1 / expt(x, -i) ENDIF;

```

Listing 25: Specification of the  $\wedge$  operator

```

+(x: valid_measurement,
  y: {m: valid_measurement | unit_match?(x, m)}):
  {m: valid_measurement | unit_match?(x, m)} =
  IF preferred_system?(x, y) THEN
    x WITH [ `value := x`value + y`value]
  ELSE
    y WITH [ `value := x`value + y`value]
  ENDIF

```

Listing 26: Specification of the addition operator

### 5.2.4 Comparisons

Comparison operations (e.g.,  $<$ ) are defined similarly to how they are defined in Subsection 4.2, except that these operations rely on the `comparable?` predicate as defined in this section.

## 5.3 Analysis

The system-field library prevents accidental mixtures of systems of units (such as befell the Mars Climate Orbiter [13]). Conversions from one system of units to another require their own theory and must be *explicitly* defined as transmutations as shown in Listing 27.

```

transmutation: TYPE =
  [#
    to_factor:
      {n: nzmeasurement | explicit_system?(n)},
    from_factor:
      {n: nzmeasurement | explicit_system?(n) AND
        to_factor`system /= n`system}
  #];

```

Listing 27: Specification of the transmutation type

A consequence of the system-field library being more rigorous than the system-template library is that far more TCCs are generated, and these TCCs are often more complex.

## 6 Discussion

### 6.1 Comparison of Libraries

The four primary objectives in creating the libraries are:

1. Soundness
2. Ease of use

3. Ease of readability
4. Effect on provability

### 6.1.1 Soundness

Both libraries are logically sound. However, the system-field library prevents the accidental combination of systems of units, while the system-templated library does not. Thus, for specifications where implicit combinations of units from different systems should not be allowed, the system-templated library may allow specifications that violate this requirement. For specifications where implicit conversions from one system of units to another *are* allowed, the system-templated library performs the necessary unit checking to ensure units are consistently used, since units are defined in this library so that conversions happen automatically per their `scaling` field.

### 6.1.2 Ease of Use and Ease of Readability

Both libraries are virtually identical in terms of ease of use and readability. The libraries only differ in this objective for specifications where different systems of units are combined. In specifications with mixed systems of units, in the system-field library conversions from one system to another must be made explicitly, making this library slightly harder to use than the system-templated library. Which library is more readable for specifications with mixed systems of units depends on preferences for explicitness or implicitness.

### 6.1.3 Effect on Provability

In the system-templated library, all measurements are valid and compatible with respect to multiplication. This results in fewer TCCs and often simpler proofs. While the system-field library supports rigorous analysis of units and eliminates the possibility of multiplying `m * ft`, its use often results in complex type-correctness conditions (TCCs) and increases the difficulty of proving theorems, compared to the system-templated library. In most cases, these TCCs are automatically proven by the built-in prover strategies, but occasionally these TCCs must be manually proven. When TCCs must be manually proven, the proofs are usually simple to complete, with the `(grind)` strategy frequently sufficing.

## 6.2 Related Work

Previous work has been done to use dimensional analysis in specifications, both in Z [3] and Simulink [9, 11].

In the Z implementation of Hayes and Mahoney [3], a measurement is defined using the  $\odot$  operator between a value and a unit. To represent a speed of 5 m/s, they would use the notation  $5 \odot L \cdot (T \uparrow -1)$ . Systems of measurement are embedded in the definition of L and T. Scales are not part of measurement, so the unit mm would be represented as  $0.001 \odot L$ . This implementation does not support exponentiation to non-integer values.

In DimSim, dimensions are defined by annotating specific Simulink blocks (e.g., source/sink blocks) using a form of vector notation [9]. In this notation, a speed measurement would be annotated with  $\langle L = 1, M = 0, T = -1 \rangle$ . The system of measurement is implicit (not specified), and scaling is not a part of this annotation, so it does not detect mistakes such as adding centimeters to meters.

SimCheck uses a similar approach to DimSim with annotations being associated with specific Simulink blocks using a form of vector notation [11]. For example, to indicate that block `RelativeSpeed` is a speed, an annotation block would be created in the model containing the statement `unit(RelativeSpeed) = [1, 0, -1, 0, 0, 0, 0]`. As with DimSim, the system of measurement is implicit (not specified), and scaling is not a part of this annotation.

While these approaches are useful, we believe that our approach of indicating a speed of 5 m/s by the notation of  $5 * m/s$  is more intuitive to read and write. Our approach also makes the system of measurement explicit, allowing our libraries to detect mistakes caused by mixing systems of measurement. Finally, we include a scale factor, so that both cm and m can be used in a specification and mistakes can be found if the two are used incorrectly.

Dimensional and unit analysis is also supported in a variety of programming languages (e.g., F# [7], Fortress [1], GHC Haskell [2], and Java [10]). Of particular interest is a prototype application from Xiang et al. that binds real-world types (including measurements) associated with cyber-physical systems with the supporting software [12, 8].

## 7 Conclusion

### 7.1 Future Work

The two libraries discussed in this paper focus on value, dimensionality, scaling, and system of measurement. However, there are other properties of physical phenomena that could be represented, including uncertainty of measurements, latency, and reference frames[4]. Future work will research means to incorporate additional real-world information into PVS in a manner that is easy to read, easy to use, and practical for proving important properties in PVS.

### 7.2 Summary

Mistakes in specifications arising from the improper representation of physical phenomena can mask fundamental flaws in those specifications. Identifying these mistakes during the specification phase of development can reduce the total cost of development. The PVS libraries discussed here provide a manner to accurately model dimensionality, scaling, and system of measurement in cyber-physical systems in a manner that is easy to read, easy to use, and without significant proof overhead. These libraries can be incorporated in automatic translations from Simulink models into the PVS specification language to allow useful properties to be proven about the Simulink models[6].

The measurement libraries are part of an open source project that is publicly available on GitHub at <https://github.com/DependableComputing/dcpvslib>.

## Acknowledgment

The authors thank Toyota InfoTechnology Center, USA and Toyota Motor Corporation for their support. This work was funded in part by USAF AFLR/RQQA contract FA8650-14-C-2528.

## References

- [1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukeyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The fortress language specification. 2007.

- [2] Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in ghc haskell. In *ACM SIGPLAN Notices*, volume 50, pages 11–22. ACM, 2015.
- [3] Ian J Hayes and Brendan P Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7(3):329–347, 1995.
- [4] Ashlie B. Hocking. Real-world contracts - rich semantics for formal interfaces, June 2015.
- [5] Ashlie B. Hocking. Dependable Computing PVS libraries, 2017.
- [6] Ashlie B Hocking, M Anthony Aiello, and John C Knight. Static analysis of physical properties in Simulink models. In *Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on*, pages 8–11. IEEE, 2015.
- [7] Andrew Kennedy. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*, pages 268–305. Springer, 2010.
- [8] John Knight, Jian Xiang, and Kevin Sullivan. A rigorous definition of cyber-physical systems. *Trustworthy Cyber-Physical Systems Engineering*, page 47, 2016.
- [9] Sam Owre, Indranil Saha, and Natarajan Shankar. Automatic dimensional analysis of cyber-physical systems. In *FM 2012: Formal Methods*, pages 356–371. Springer, 2012.
- [10] JSR-108 project. Unit (Java Units API), 2004.
- [11] Pritam Roy and Natarajan Shankar. SimCheck: a contract type system for Simulink. *Innovations in Systems and Software Engineering*, 7(2):73–83, 2011.
- [12] Jian Xiang, John Knight, and Kevin Sullivan. Real-world types and their application. In *International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2015.
- [13] T. Young, J. Arnold, T. Brackey, M. Carr, D. Dwoyer, R. Fogleman, R. Jacobson, H. Kottler, P. Lyman, and J. Maguire. Mars Program Independent Assessment Team Report. *NASA STI/Recon Technical Report N*, page 32462, March 2000.