Improved Unit Inference and Checking in Modelica

Hans Olsson[1]

[1]Dassault Systemes AB, Sweden, hans.olsson@3ds.com

Abstract

This paper will present a new unified algorithm for unit checking and inference, and showing the benefits for various libraries.

The Modelica Language supports declaring units for variables using the SI-standard. This allows dimensional checking to detect possible errors in equations. The units for variables make it easier to interpret, input and plot their values. When we infer the unit of a variable we get the same benefits also for variables without a declared unit. We will use unit inference and checking for the combination, even if the check is primarily a dimensional check.

Both dimensional checking and unit inference are already implemented in several Modelica tools, but not consistently. The original motivation for this paper was to understand the different approaches, and demystify the unit handling with the goal of making it more available. Based on that understanding, this paper will also present a new unified algorithm combining the different strengths, and showing the results for various libraries.

Keywords: unit, Modelica, Hindley-Milner

1 Unit background

This will explain the various considerations for the units of variables and unit equality in expressions.

1.1 Allowed units

The stated goal in Modelica is that variables should use SI-units – for the unit-attribute, the displayUnit-attribute is not considered in this paper. The SI-system (BIPM 2019) is based on seven base quantities, and every quantity is a product of them raised to various powers (called the dimension of the quantity). If all powers are zero it is called dimensionless. The seven base quantities correspond to the seven base units of the SI-system, and all units are directly expressed – without any conversion factors. This eliminates some causes of errors, provided only SI-base-units are used. However, when equations contain different prefixed SI-units (e.g., mm instead of m or g instead of kg[1]), or SI-acceptable units like bar or kWh the dimensional analysis does not suffice, and that is an important aspect that will be considered. Additionally to directly quote (BIPM 2019)

> "In practice, with certain quantities, preference is given to the use of certain special unit names to facilitate the distinction between different quantities having the same dimension. ... Even though torque has the same dimension as energy (SI unit joule), the joule is never used for expressing torque."

Plane angles (with unit radian "rad") and solid angles (with unit steradian "sr") are a supplementary units in the SI-system, and even if they are useful for human understanding they are formally dimensionless. Since radians are important we follow Mattsson and Elmqvist (2008) and the FMI-standard (see (Modelica Association 2014)) and treat them similarly as the seven base units when possible – but allow mismatches.

It has also been proposed to add new orthogonal non-physical base quantities like currency to Modelica; the approaches in this paper support them as well.

1.2 Unit Strictness

This paper builds on the proposal from Casella (2016) for unit checking. The basic idea is that literal values inside multiplicative expressions are dimensionless, while literal values on their own can have any unit. That has been found to be a pragmatic solution that allows checking and infering units in a large number of existing Modelica models without any changes. Consider

```
parameter SI.Capacitance C=2; // Ok
SI.Energy E1=C*v^2/2; // Ok
SI.Energy E2=C*v*4; // Error
SI.Voltage v;
```

On the first line `2` gets the desired unit, but on the next line two `2` are dimensionless. The `E2` line is rejected even if we could find a unit for `4` that makes it correct.

Note that for this proposal the equal sign does not fully represent equality, since replacing `C` by `2` in another equation leads to unit errors – or incorrect unit inference. In particular this is an issue if the model was originally constructed by manually substituting expressions based on equality.

The approaches described in this paper work with other rules as well. Alternatives include allowing numeric literals to have any unit everywhere except for `1/x` see (Mattsson and Elmqvist 2008), the same without the exception, or only allowing the zero literal to have any dimension see (Kennedy 1996). The future work will contain some proposals to modify the rules, both to strengthen them for some common cases, and weaken them for experimentally derived equations.

[1]Remember that the base-unit for mass is kg

1.3 Arrays

Most arrays are homogenous, i.e., all of their elements have the same unit. However, there are some arrays that are heterogenous – in particular the state of the transfer-function (each element is the derivative of the previous one) and polar coordinates as an array of two elements (magnitude and angle). Thus this paper will allow arrays to be heterogenous, but try to infer the common unit if it exists and/or infer units for specific elements (for array elements with literal subscripts – they are seen as scalars).

One can consider variants of this, one would be to enforce that all arrays are homogenous, and another would be to perform all unit inference on the scalar level. Neither extreme seems acceptable, but the first approach will be discussed more later.

Instead the solution is to view an array built as `C=[A,B]` as only having a unit if it is homogenous (the implications will be discusssed later). Similar considerations apply to subscripted elements of arrays.

The subscripts themselves are integers and should thus be dimensionless.

1.4 Unit for variables

The general idea is that a variable has a statically determined unit, or is free to have any unit. That may seem obvious, but apart from heterogenous arrays some libraries (Zimmer, Meißner, and Weber 2022) have variables getting different units depending on an evaluable parameter. Another case are constants such as `eps` – that are treated like small literal values. Trying to deduce a unit would be especially problematic for package constants, because the same constant can be used in unrelated parts of a model.

For algorithms (in particular in functions) one could imagine that one variable is re-used to store values with different units – we will consider that an error. Other common cases where one variable have multiple units are s-parameterizations, where positive and negative values have different units – that can be handled by making both cases dimensionless in the model, see (Mattsson and Elmqvist 2008). An important consideration is that variables are free to not have a unit, this allows building blocks that can be re-used with different units.

2 Approaches to unit inference and checking

There are multiple approaches to unit inference and checking in various programming languages, both related to the nature of the language and to the goals of the check.

2.1 Simple unit inference and checking

In traditional imperative programming languages it is straightforward to propagate unit forwards from variables, so that the unit of `z=x*y` gets the unit of `x` multiplied by the unit of `y`, and addition will just check that the two terms have the same unit, and give the result that unit. This has been implemented multiple times, see (Hilfinger 1998) for an early variant for ADA.

This propagation up-wards is insufficient in an equation-oriented language such as Modelica. Dymola has traditionally extended this with also propagating the unit down-wards, i.e., if we know the units of `z` and `x` in `z=x*y` we conclude the unit of `y` is the unit of `z` divided by the unit `x`. The basic framework for unit checking in Dymola is described by Mattsson and Elmqvist (2008) including down-ward propagation. The unit-checking in Dymola has after that paper built on that framework and can be configured in different ways. In this paper the rules for literals (literals have any unit, except for `1/x` where `1` is dimensionless) are replaced by the stricter rules from Casella (2016) that only allow literals to have any unit in additive contexts so that we can for `z=3*x+2` propagate the unit from `x` to `z`. Specifically the multiplicative literal 3 is dimensionless (allowing propagating the unit), but the additive literal 2 has any unit (avoiding a unit error).

A practical application of this can be seen in the CoupledClutches example for `sin1`, which simplified has:

```
  RealInput torque.tau(unit="N.m");
equation
  sin1.y = sin1.offset + sin1.amplitude*sin
      (2*pi*sin1.f*time);
  connect(sin1.y, torque.tau);
```

Here we propagate “N.m” through the connection and then down-wards to `sin1.offset` and `sin1.amplitude` (knowing that the output of `sin` is dimensionless).

This approach needs to be iterated (normally just a handful of times), and at first seems ad-hoc.

It does, however, work fairly well in practice. The benefits of the approach are:

- The results are understandable, for each inferred variable one can directly see which equation that the unit originates from. That is useful for analyzing errors for unit checking where having a clear understanding of how the various variables got their unit helps in finding the error.
- For unit inference it can preserve the exact units used; so if a variable has unit “N.m” variables that are equal to it can also get unit “N.m” instead of “kg.m2.s-2” or the even more confusing “J”. This is also important when not using SI-base-units, e.g., “mol/l”. Apart from "N.m" being required by the SI-standard for torque this also works together with non-SI displayUnits (suggesting “kcal” or “kWh” as display unit for a torque is not good).
- Simple and consistent handling of arrays variables:
  - We can treat homogenous arrays as scalars, and if we infer a unit for an array that is valid for the entire array. Conversely, if we know that array is heterogenous we can treat it as having unknown unit (see (Mattsson and Elmqvist 2008)).

DOI 10.3384/ecp21817

  - It is straightforward to limit the unit inference to going in one direction (conditional unit assignments). An example where this is useful is an equation involving array indexing. If the entire array has one unit that should also be the unit for the array element, but in order to support heterogenous arrays the inference cannot go from unit of an array element to unit of the entire array.
  - It is also straightforward to handle a `C=[A,B]` such that if we have a unit for `C` we can infer the unit for `A` and `B`, and if `A`, `B` both have a unit and it is the same, we can infer that `C` has the same unit. It is deliberate that if only `A` has a unit we do *not* assume that `B` (and `C`) must have that unit as well.
  - Array elements with literal indicies can be treated as scalars.

That it works for arrays is not limited to simple cases like adding two vectors with units, but also extends to scalar products, cross products, and matrix multiplications.

2.2 Hindley-Milner unit inference

The Hindley-Milner (or Damas–Hindley–Milner) unit inference traces it origins to the Hindley-Milner algorithm for type inference, and is presented as Damas-Milner by Kennedy (1996) (an alternative reference is his PhD thesis from the same year). It has previously been discussed for Modelica by Broman, Aronsson, and Fritzson (2008) and proposed by Lambert and Tidefelt (2024). We will go through it in detail, but it basically collects all unit-equations and in one sense solves them one by one and then substitutes the result in the other equations (and substitutions), until all of the unit equations have been handled.

The benefits of this approach are:

- Uses all of the unit information and nothing more, i.e., it is complete and consistent.
- Known convergence and practically linear.

2.2.1 Details

The first step in this approach is to collect all of the unit equations as equations.

Consider the model

```
model Elec
  Real pot1(unit="V"), pot2(unit="V");
  Real v, i;
  Real R(unit="Ohm");
equation
  v = i * R;
  v = pot2 - pot1;
end Elec;
```

The variable `pot1` is called p_1 below, and similarly for `pot2`. This gives the unit-equations including substitutions for known units (we use `u[v]` for the unit of `v` as a symbolic value, and reserve `v.unit` for the corresponding string).

$$u[p_1] \to V = kg \cdot m^2 \cdot s^{-3} \tag{1}$$
$$u[R] \to \Omega = V \cdot A^{-1} = kg \cdot m^2 \cdot s^{-3} \cdot A^{-1} \tag{2}$$
$$u[v] = u[i] \cdot u[R] \tag{3}$$
$$u[v] = u[p_1] \tag{4}$$
$$u[v] = u[p_2] \tag{5}$$

In practice it makes sense to have all unit-equations in the form $1 = \ldots$ – to ensure that they are normalized, $u[x]$ is the unit or dimensionality of the variable `x`, and substitutions are used for *solved* equations. The substitutions are seen as solved even if they may depend on other (unknown) units. Variables with unit-modifiers are also represented as substitutions.

Another example, which will be discussed later, is:

```
model SecondOrder
  input Real u;
  parameter Real w, D, k(unit="1")=1;
  Real y(unit="1"), yd;
equation
  der(yd) = w*(w*(k*u-y)-2*D*yd);
  yd = der(y);
end SecondOrder;
```

The declaration of k gives:

$$u[k] \to 1 \tag{6}$$

The declaration of y gives:

$$u[y] \to 1 \tag{7}$$

The subexpression `k*u-y` gives:

$$u[y] = u[k] \cdot u[u] \tag{8}$$

Introduce α_1`=w*(k*u-y)` giving:

$$u[\alpha_1] = u[w] \cdot u[y] \tag{9}$$

The subexpression α_1`-2*D*yd` gives

$$u[\alpha_1] = u[D] \cdot u[yd] \cdot \tag{10}$$

The equation `der(yd)=w*(`$\alpha_1 - \ldots$`)` gives:

$$u[yd] \cdot s^{-1} = u[w] \cdot u[\alpha_1] \cdot \tag{11}$$

The equation `yd=der(y)` gives:

$$u[yd] = u[y] \cdot s^{-1} \tag{12}$$

Equations 8 to 11 all derive from one Modelica equation. The variable α_1 is only introduced as part of unit-checking and may more accurately be described as only introducing $u[\alpha_1]$ corresponding to the unit of `w*(k*u-y)`, but not α_1 itself. Using such intermediate variables makes the construction of the unit-equations more straightforward even for complicated equations. Additionally they are needed in the proposed combined algorithm for some cases. They can trivially be excluded when reporting the result of the unit inference.

One could even introduce $\alpha_2 = k \cdot u - y$ replacing (8) by $u[\alpha_2] = u[y]$ and $u[\alpha_2] = u[k] \cdot u[u]$, and for (9) use $u[\alpha_1] =$

$u[w] \cdot u[\alpha_2]$. However, since $u[\alpha_2] = u[y]$, one can use $u[y]$ as above.

This example will be used to illustrate that Hindley-Milner can give additional units compared to the simple algorithm (which cannot find the units for `D` and `w`).

If performed naively a second intermediate step (normally included in the first step) solves some issues:

- Combine exponents for the same variable, which may even cause a variable to disappear from the unit equation (if it appears on both sides of the equation with equal exponents).
- To support `sqrt` and `nthRoot` the original unit-equations may contain rational exponents for the unit-variables, to simplify the analysis each unit-equation is raised to a suitable power to get integer coefficients (either for just the unit-variables or also extended to known units).

The third step is to take these unit-equations (in any order) and for each find a unit-variable with the exponent that is smallest in magnitude. As listed by Kennedy (1996) there are number of cases:

- There is no unit-variable, just check that equation is unit-correct.
- There is only one unit-variable remaining, find its unit and substitute its unit in the other unit-equations and substitutions.
- There are multiple unit-variables, and the smallest exponent is +/-1 or more generally the smallest exponent divides the exponents of all other unit-variables. In this case solve for such a unit-variable (as a product of fixed units and a product of other unit-variables).
- The remaining gcd-case where we introduce a new unit-variable by dividing the other exponent as much as possible, and forming a new unit-equation with the remainders, and then iterate on this unit-equation until we get to one of the other cases. This will convergence in a finite number of steps and can be seen as way of finding the greatest common divisor of those exponents, see Appendix B for details.

As an alternative to the last step one could allow rational numbers as exponents for variables, and use the penultimate step in all remaining cases (Lambert and Tidefelt 2024). The main disadvantage would be that it spreads the rational numbers.

The only really difficult part of this algorithm is understanding that it actually works, and implementing the gcd-case (or using rational numbers). Actually working mean that the algorithm finishes and it either produces a unit error, or the same unit inference (independent of the order of the unit-equations) and that the results are consistent. The gcd-case is the only concern for showing that the algorithm finishes, and that relies on the convergence of the Euclidean algorithm for computing the gcd of integers. Consistency also imply that if we add any of the inferred units for the variables to the unit-equations and re-run the algorithm it will produce the same result.

For the Elec-case the most efficient way would be to solve the second equation for $u[v]$ and then the first equation for $u[i]$, but we will show that we can solve them in the given order, which means that there is no need to describe an optimal order:

substituting the known units gives

$$u[v] = u[i] \cdot \Omega \tag{13}$$

$$u[v] = V \tag{14}$$

solving the first (13) for $u[v]$ gives

$$u[v] \to u[i] \cdot kg \cdot m^2 \cdot s^{-3} \cdot A^{-1} \tag{15}$$

substitution this in (14) gives

$$1 = u[i]^{-1} \frac{kg \cdot m^2 \cdot s^{-3} \cdot A^{-1}}{kg \cdot m^2 \cdot s^{-3}} = u[i]^{-1} \cdot A \tag{16}$$

solving this gives

$$u[i] \to A \tag{17}$$

substitution this in (15) gives

$$u[v] \to A \cdot kg \cdot m^2 \cdot s^{-3} \cdot A^{-1} = kg \cdot m^2 \cdot s^{-3} \tag{18}$$

and subsitution in the two remaining equations

$$u[p_1] \to u[v] = kg \cdot m^2 \cdot s^{-3} \tag{19}$$

$$u[p_2] \to u[v] = kg \cdot m^2 \cdot s^{-3} \tag{20}$$

And thus `v.unit="kg.m2.s-3"` (or `v.unit="V"`), and `i.unit="A"`. As noted above solving (14) from $u[v]$ and then (13) for $u[i]$ is an alternative giving the same result.

For SecondOrder we can start by handling the trivial cases to get:

$$u[k] \to 1 \tag{21}$$

$$u[y] \to 1 \tag{22}$$

$$u[u] \to 1 \tag{23}$$

$$u[yd] \to s^{-1} \tag{24}$$

$$u[\alpha_1] \to u[w] \tag{25}$$

and the non-trivial equations (10) and (11) gives:

$$u[w] = u[D] \cdot s^{-1} \tag{26}$$

$$s^{-2} = u[w]^2 \tag{27}$$

the non-trivial part (26) becomes

$$u[w] \to u[D] \cdot s^{-1} \tag{28}$$

substitution in (27) and simplifying gives

$$1 = u[D]^2 \tag{29}$$

solving (29) and substituting in (28) gives

$$u[D] \to 1 \tag{30}$$

$$u[w] \to s^{-1} \tag{31}$$

DOI 10.3384/ecp21817

The key part is that the substitutions combine `D` and `w` into one equation that can be solved, something that the simple algorithm cannot do. In particular the simple algorithm does not have the unit equation (27) due to how the equations are written, but would have it for the equivalent equation `der(yd)= w^2*(k*u-y)-2*w*D*yd;`

If we remove the unit for `y` we would not infer units for `yd`, `u`, and α_1, but the inferred units for `D` and `w` would be unchanged.

An equivalent approach is to solve integer equations for the units (this is a traditional approach in dimensional analysis - and also seen in this context by Karr and Loveman III (1978), for the connection see (Kennedy 1996)) -– those integer equations correspond to taking the logarithm of the unit-equations used by Hindley-Milner. It is important to note that the equations are integer equations and treated as such -– using the same approach for solving systems of equations with real coefficients is not numerically robust.

For this case the left-hand side of the integer equations are:

$$\begin{pmatrix} +1 & & & & & & \\ & +1 & & & & & \\ +1 & -1 & +1 & & & & \\ & -1 & & +1 & & & \\ & +1 & & & +1 & -1 & \\ & & & -1 & +1 & +1 & \\ & & & +1 & & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \log(u[k]) \\ \log(u[y]) \\ \log(u[u]) \\ \log(u[yd]) \\ \log(u[w]) \\ \log(u[\alpha_1]) \\ \log(u[D]) \end{pmatrix} \tag{32}$$

They have been arranged to be in block-lower-triangular form, where the non-trivial block corresponds to solving for a combination of variables.

3 Comparison of the approaches

We will both investigate the relationship between the two approaches in theory, and then practically investigate it for the Modelica Standard Library.

3.1 Theoretical considerations

If we only consider scalars then the Hindley-Milner approach can infer more units, and thus also detect more errors.

There are two reasons for this:

- The combination of exponents for variables can infer the unit of `x` from the unit of `x*x` and to all variants of this (the simple algorithms handles `x^2` and extending it to detect `x*x` is trivial). Disappearing variables allows deducing the unit of `T` from `T*der(x)=x` (`T` has unit "s"), and also from `r*x+x` (`r` has unit "1"); without requiring a unit for `x`. Basically we for the first case get $u[T] \cdot u[x] s^{-1} = u[x]$, and divide both sides by $u[x]$ giving $u[T] = s$.

- Substitution means that if we have a damping factor `D` and a frequency `w` and the equations `der(yd)= w*(w*(k*u-y)-2*D*yd)` and `yd=der(y)` we can find the unit for both `D` and `w`.

The similarity between the two approaches is that by a suitable ordering of the unit-equations we can view the simple inference as prematurely stopping the Hindley-Milner algorithm. Basically we stop before we get to any of the problematic cases where we would *need to* substitute a unit-variable depending on another unknown unit-variable, or combine unit-exponents. This has some implications consequences:

- The simple algorithm gives a consistent sub-set; not a different result.

- By continuing with Hindley-Milner after running the simple algorithm we clearly see which additional unit inferences (and possible errors) it gives.

There are some caveats here:

- It is only the case for a suitable ordering, the Hindley-Milner can process the unit-equations in any order, and thus first find that the unit of `z` depends on the unit of `x`, and after substituting this solve another equation giving the unit for `x`.

- It is only the case if the unit-equations are created in a suitable way, specifically to handle `z=a*b+y` the simple algorithm would be stuck if we add the equations $u[z] = u[a] \cdot u[b]$ and $u[a] \cdot u[b] = u[y]$ instead of $u[z] = u[a] \cdot u[b]$ and $u[z] = u[y]$. A solution is to introduce a new unit-variable α_1 and $u[\alpha_1] = u[z]$, $u[\alpha_1] = u[a] \cdot u[b]$, $u[\alpha_1] = u[y]$, since it ensures that as soon one of the three unknowns get a unit it can be propagated to the other two (in this case we can skip the extra variable and reuse `z` or `y`, but in general all of them could be complicated expressions). Hindley-Milner can handle these cases without new variables.

- It also requires that we don't convert known units to base-units in Hindley-Milner during that part (in order to get the benefit of the original units). In theory this could be disastrous for performance, but in practice it does not seem problematic and if it is a problem it could be limited to units with a few factors.

- Combining arrays and Hindley-Milner causes problems. This will be discussed later.

Keeping the original units (i.e., not converting to base-units) in the normal Hindley-Milner would be more complicated due to the repeated substitutions, that problem practically disappears due to the suitable ordering.

For unit-errors keeping the original units allows the error message to include both the original units and the result converted to base-units; where either may be more helpful depending on the circumstances.

Additionally, this suggest that the algorithms can be unified, i.e., first running the part of Hindley-Milner corresponding to the simple algorithm – even extended to combining variables. That will automatically preserve most of the benefits of the simple algorithm (locality and understandable unit strings), and then later running the complete algorithm for the small additional gain – but restricted to scalars.

Instead of iterating all equations for the simple algorithm one could keep track of where unit-variables are used and after updating the impacted ones for each substitution check those substitutions to check whether there is at most one unknown unit in that equation. That ensures that the simple algorithm is practically linear, similarly as Hindley-Milner.

3.1.1 Array issue

Lambert and Tidefelt (2024) indicate that there are issues for arrays by limiting it to scalars, but we want to make the issues clear. The obvious issue is that conditional equalities for arrays (for subscripting and concatenation) do not naturally fit the unification based on equality.

However, they can be handled by introducing an unknown array variable (in case the array occurred inside an expression) and after substitutions check if either that array has a unit, or the other side has homogenous units and only then add it as an equality. It is not known whether this will give good results when substituting unit-variables depending on others.

There is also a worse problem that is less obvious, consider `T*der(x)=A*x;` For scalars we can see that `A.unit="1"` (regardless of the unit of `x`), and simply applying the algorithm would give the same result for a matrix `A`. However, if we have `x.unit={"m","s"}` then we should have `A.unit=["1","m/s";"s/m","1"]` (i.e., only the diagonal has unit 1). Solutions to this would be to scalarize before performing unit inference and checking, or adapting the algorithm to handle the various kinds of array equations. In practice both variants would make the implementation considerably more complicated as one needs to ensure that scalarization does not perform any simplification that would impact unit inference or checking. Neither approach has been implemented.

Note that the problem does not occur for *all* array expressions. E.g., if `A` is a scalar then the simple result is correct regardless of whether `x` is a scalar, vector, or matrix. Similarly, for `v1*x=v2*x` or `v1.*x=v2.*x` we can eliminate `x` to conclude that `v1` and `v2` have the same unit(s).

However, even if non-homogenous arrays exist in some models having an option to test enforcing homogenous array is useful — the idea would be to discover missed units, and manually add them to the model.

3.2 Practical difference

The previous section showed that it is possible to first run the simple algorithm, and then extend it to the Hindley-Milner algorithm (for scalars). During these tests we naively keep the unconditional array equations in the second step.

This means that we can isolate the impact of the advanced parts. This has been implemented and tested on the Modelica Standard Library (MSL, specifically on (Modelica Association 2025b)) and the experience was:

- The Hindley-Milner approach is actually easier to implement and also easier to get right.
- The reason is that for the simple unit inference one needs to implement unit-inference going both "up" and "down". If there are a large number of different operations that can be substantial work, and instead Hindley-Milner only needs to go "up" and collect equations in a standardized form.
- The most complicated part of implementing Hindley-Milner is the gcd-case, and that does not occur in MSL (it is unclear if it occurs for any actual models).
- No new unit-errors were reported by the algorithm compared to the simple algorithm (there are a number of known unit-issues in the library — even if they are being reduced).
- Apart from squared variables, it detected new units for variables in seven models that could not be found by the simple algorithm; three of them were useful – three of less practical use, and in one case the results were incorrect (due to how the model was constructed – it did not indicate a problem with the algorithm).

There are two kinds of squared variables, one was propagating a unit down for `V=sqrt(re*re+im*im)` for electrical models. The other occurs if we keep array equations and gives that `T` has unit "1" from `r1=transpose(T)*T*r2` (where `r1` and `r2` are positions). The latter is the only case where naively keeping array equations made any difference. In this specific case the unit is correct (and the derivation is even correct in general for this specific array expression), but given that `T` is the orthogonal transformation matrix we even know that $T^T \cdot T = I_3$, and thus that `transpose(T)*T` does not matter in that equation. This indicates that at least for MSL skipping array equations in the second part of the full algorithm has neglible impact.

The useful results were for models such as SecondOrder which has `der(yd)= w*(w*(k*u-y)-2*D*yd)`, where we can find the unit for both `D` and `w`; even if we cannot find either unit directly.

The ones of less practical use were UniformNoise from the Clocked part which has `y = u + noiseMin + (noiseMax - noiseMin)*noise;` where we see that `noise.unit="1"`. The unit for that variable is of limited use as it is a protected variable (so no impact on other variables and not intended for plotting), and `unit="1"` isn't useful for unit conversion.

DOI
10.3384/ecp21817

The one with incorrect result was the most interesting and it is the Motor model used in the R3Robot example where we have:

```
  parameter Real w=4590 "Time constant of
      motor";
  parameter Real D=0.6 "Damping constant of
       motor";
...
  Modelica.Electrical.Analog.Basic.Inductor
       La(L=(250/(2*D*w))) ...;
  Modelica.Electrical.Analog.Basic.Resistor
       Ra(R=250) ...;
  Modelica.Electrical.Analog.Basic.Resistor
       Rd2(R=100) ...;
  Modelica.Electrical.Analog.Basic.
      Capacitor C(C=0.004*D/w) ...;
```

Just applying the algorithm gives the conclusion that $u[D] \rightarrow \Omega^{-1}$ (or Siemens), and $u[w] \rightarrow s^{-1}$, which is based on assuming that literal numbers have unit "1" inside equations (but unknown unit when they appear alone). Since the unit is seemingly correct for `w` it would be tempting to consider the entire result correct. However, looking carefully shows that 0.004=1/250, and an obvious thought is that all three uses of 250 likely have unit Ohm. Testing this by replacing 0.004 by 1/250 and then replacing those three uses of 250 by a common parameter gives the result $u[D] = 1$, $u[w] = s^{-1}$; and a damping constant 0.6 (with unit "1") make perfect sense.

The SI-standard, (BIPM 2019), recommends against "s-1" due to the ambiguity of whether it is a frequency ("Hz") or angular frequency ("rad/s"), numerically they differ by a factor of 2π. However, it is not possible to resolve this ambiguity automatically and thus the non-recommended "s-1" is safest. Manual checking (and also the name `w`) shows that it is an angulary frequency, "rad/s". The extra factor "rad" may seem to be unit-incorrect, but this supplementary unit is dimensionless, and thus the capacitance and inductance are unit-correct even if they do not contain a factor "rad".

Normally the parameter with value 250 would explicitly be declared as a resistance parameter, but that could also be inferred. A pull-request has been prepared that declares `w` and `D` with proper types with units, and also add the resistance parameter; as that is a more understandable model. Just adding the resistance parameter would still require Hindley-Milner to infer the units for `w` and `D`.

3.3 Impact of unit inference and check

The new algorithm was tested on different libraries and the variables were split into different categories. Variables in common models are counted multiple times, but the special variable `time` was excluded. Only models that passed unit check were considered for the inference statistics.

Library	MSL		MSL
Version	4.0.0		3.2.3
Scalars	336899		329763
Declared	238268	(70.7%)	239798
Simple	9171	(2.7%)	9325
Advanced	44	(0.01%)	39
Arrays	73553		55222
Homogenous	57312	(77.9%)	38384
Heterogenous	191	(0.3%)	308
Simple	4869	(6.6%)	1035
Advanced	0		0
Elements	40	(0.05%)	55
Tot. Classes	6299		5954
Inference	1083		931
Errors	67		147
Adv. Errors	0		2

And some smaller libraries for comparison.

Library	ThermoFluid-	VehicleInterfaces
Version	Stream 1.1.0	2.0.1
Scalars	43163	9692
Declared	33582	4287
Simple	1268	143
Advanced	0	0
Arrays	3198	10579
Homogenous	2774	8508
Heterogenous	0	0
Simple	15	712
Advanced	0	1
Elements	0	0
Tot. Classes	2156	252
Inference	106	28
Errors	40	14
Adv. Errors	0	0

The "simple" are scalar and homogenous arrays inferred by the simple algorithm. The "advanced" refer to the additional ones found by the algorithm proposed in this paper. For MSL the "advanced" cases are the previously mentioned squared variables in electrical models, and the seven additional cases – which are counted multiple times due to model re-use. The "elements" refer to arrays where units of some elements are inferred by the simple algorithm. The "declared" and "homogenous" and "heterogenous" are variables that are declared with units.

Although counting all variables the same is oversimplified it still gives an indication of the impact. The percentages do not fully capture the advantages, e.g., the simple algorithm only inferred the unit for 6.6% of the arrays in MSL 4.0.0, but that represents 30% of the arrays that lacked a declared unit.

The number of arrays where elements have different units (both declared as "heterogenous" and deduced as "elements") underestimates their already small number. The reason is that algorithm is not designed to find them, and e.g., state-space vectors are not detected as having elements with different units.

The "tot. class" include a large number of documentation classes, interface classes (where no inference is pos-

sible), and other classes without units and thus the classes with unit inference is more interesting. The "errors" are the number of classes containing errors, where a single error may be reported for multiple classes. As the unit-proposal is yet to be approved, the "errors" do not necessarily indicate quality problems.

MSL 3.2.1 had two models with advanced errors, they were both due to the gain `k` having unit "1" in the PI-block, which is used in the Controller in the NoiseExamples. That has later been removed in general, and did not indicate any error in the tested model. Note that the number of classes, including the classes with inferred units have increased in the later MSL version – while the number of errors has decreased.

4 Combined algorithm

The array issues imply that a combined algorithm would be to run Hindley-Milner corresponding to the simple algorithm (including arrays — both simple array equations and one-sided; and without combining unit-exponents and substituting unknown units), then combine exponents and run the full Hindley-Milner on the remaining unit equations only involving *scalar* variables. We would generally propose this as it is simple to explain and gives reasonable results.

However, a more detailed analysis shows that also simple equations involving arrays can be kept in the second phase, as long as one only combines exponents for scalars and only uses substitutions with scalars in the right-hand-side. (The reason is that if an array has a unit depending only on the unit of scalar variables multiplied by a fixed unit then it must be a homogenous array.) One could even extend this from only scalars to scalars *and homogenous arrays*, where operations such as `sum(v)`, `max(v)`, or `v*u` (where `u` is homogenous) imply that `v` is homogenous.

As the practical tests only found one case with any benefit from naively keeping array equations the difference between these variants will be minor.

One can construct cases where the difference between these algorithm matters, e.g.,

```
  Real x;
  Real v[3];
  Real w[3](each unit="m");
equation
  v*v=1; // Normalize
  v*x=w;
  ...
```

Considering equations involving arrays is needed to find the unit for `v` and `x`. Without any special logic for `v*v`, we can solve the second equation for $u[v] \to m/u[x]$, and substitute that in the first equation to give $u[x] \to m$ and thus $u[v] \to 1$.

5 Future Work

The most important future work for the Modelica Language is to get an agreement on the unit rules. For tools the important part is implementing these rules.

For libraries (not only MSL) the first part is to remove any unit errors and in some cases incorrect units. A particular consideration is to use rational exponents consistently, both in units-string (added by Modelica Association (2025a)) and in models. For models this implies one should use `sqrt(x)` instead of `x^0.5` and `nthRoot(V, 3)` instead of `V^(1/3)` (also introduced by Modelica Association (2025a)). Automatically handling some common rational exponents (with diagnostics) is possible and helpful for users, but using `sqrt` and `nthRoot` more clearly shows the intent.

Unit handling in equations involving arrays could be investigated further. An obvious improvement would be to prove that some operations require homogenous input (such as `sum` and `cross`) and view their inputs as homogenous *everywhere*.

Combining this approach with the analysis on subcomponents for partial result as by Broman, Aronsson, and Fritzson (2008) would be interesting. Having the possibility to store the inferred units in the model seems like a natural possibility for the future, but it seems redundant and if too automated it may include dubious units like “s-1” instead of either “rad/s” or “Hz”.

5.1 Proposed Extensions

Adding a generic type for the dimensionless case to `Modelica.Units.SI` would make it easy to clarify the many cases where a variable is known to be dimensionless (there are already a number of types for *specific* dimensionless quantities).

Having the possibility to disable unit-checking for a specific equation, or all equations in a model has been found useful. That basically means that no unit-equations are added for the corresponding equations. Testing of the Buildings 10.0.0 library, (Wetter et al. 2014), similarly revealed a need to disable the the new rules for literals on the package level.

For converting between different prefixed SI-units, or non-SI-unit it would be beneficial to have a function `UnitConvert(a, fromUnit, toUnit, factor)` that just returns `a*factor`, that should be the unit-conversion factor from one unit to another. (Also proposed by Lambert and Tidefelt (2024)) Having a special name it clarifies the intent of convertering between the units, and including the units and the factor it is trivial to implement for tools that don’t implement units – while still allowing tools to verify that the factor is correct. The *intent* is important as one cannot assume any parameter with such a unit is a conversion – e.g., an up-hill road could have slope of 100 mm per m.

6 Conclusions

This paper shows how to unify the simple unit-algorithm from Mattsson and Elmqvist (2008) traditionally implemented in Dymola with Hindley-Milner unit-inference. It shows that this will give the benefits of both – preserving understandable units and locality for the large num-

DOI 10.3384/ecp21817

ber of simple cases – including arrays, while also getting units for all the remaining scalar cases. The understandable units make it easy to input and interpret values for variables without a declared units, including switching to compatible units for display. Any errors will also (with rare exceptions) be localized to specific equations, where the units for all involved variables are easy to trace. Even if the number of additional variables with inferred units is small, it is still worth in terms of providing a stronger theoretical basis – especially considering the effort.

Based on experience in Dymola it is even simpler to implement than the complete simple unit-inference. The algorithm has been test-implemented in Dymola 2025x Refresh 1, and will also part of 3D Experience Platform 2026x.

It also shows that only implementing a conventional Hindley-Milner approach would require more care, both to get good unit strings and either not work for arrays or require scalarization for arrays (without any clear benefit).

Acknowledgements

This work has been supported by VINNOVA (grant number 2023-00969, OpenScaling). The authors would also like to thank the Modelica Language group for pushing improvements for units.

References

BIPM (2019). *Le Système international d'unités / The International System of Units ('The SI Brochure')*. Ninth. Bureau international des poids et mesures. ISBN: 978-92-822-2272-0. URL: http://www.bipm.org/en/si/si_brochure/.

Broman, David, Peter Aronsson, and Peter Fritzson (2008). "Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica". In: *Modelica 2008*. Vol. 2855. International Modelica Conference. The Modelica Association, pp. 134–155. URL: https://modelica.org/events/conference2008/sessions/session1a1.pdf.

Casella, Francesco (2016-12). *MCP-0027 Units of Literal Constants*. Tech. rep. Cremona: Modelica Association. URL: https://github.com/modelica/ModelicaSpecification/issues/2127.

Hilfinger, Paul N. (1998). "An ADA Package for Dimensional Analysis". In: *ACM Transactions on Programming Languages and Systems* 10.2, pp. 189–203. DOI: 10.1145/42190.42346.

Karr, Michael and David B. Loveman III (1978). "Incorporation of Units into Programming Languages". In: *Communications of the ACM* 21.5. DOI: 10.1145/359488.359501.

Kennedy, Andrew John (1996-04). *Programming languages and dimensions*. Tech. rep. Cambridge: University of Cambridge Computer Laboratory. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf.

Lambert, Quentin and Henrik Tidefelt (2024-03). *Applying Hindley-Milner to unit-checking*. Tech. rep. Wolfram System Modeler: Modelica Association. URL: https://github.com/modelica/ModelicaSpecification/pull/3491.

Mattsson, Sven Erik and Hilding Elmqvist (2008). "Unit Checking and Quantity Conservation". In: *Modelica 2008*. Vol. 2855. International Modelica Conference. The Modelica Association, pp. 13–20. URL: https://modelica.org/events/conference2008/sessions/session1a2.pdf.

Modelica Association (2014-07). *Functional Mock-up Interface for Model Exchange and Co-Simulation Version 2.0*. Tech. rep. Linköping: Modelica Association. URL: https://fmi-standard.org.

Modelica Association (2025a-02). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.7 dev*. Tech. rep. Linköping: Modelica Association. URL: https://specification.modelica.org/master/MLS.pdf.

Modelica Association (2025b-02). *Modelica Standard Library*. Tech. rep. Linköping: Modelica Association. URL: https://github.com/modelica/ModelicaStandardLibrary/commit/7c4858b675d2442cd34c99047ca9fb672083e11a.

Wetter, Michael et al. (2014). "Modelica Buildings library". In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. DOI: 10.1080/19401493.2013.765506.

Zimmer, Dirk, Michael Meißner, and Niels Weber (2022). "The DLR ThermoFluid Stream Library". In: *Electronics* 11.22. ISSN: 2079-9292. DOI: 10.3390/electronics11223790.

A All used unit rules

- Addition, $u[z] = u[x], u[y] = u[x]$ for `z = x + y`, `z = x - y`, `z = x .+ y`, `z = x .- y`, `z = min(x, y)`, `z = max(x, y)`, `z=rem(x, y)`[2], `z = mod(x, y)`[2], and `z = homotopy(x, y)`[2].
- Multiplication[3], $u[z] = u[x] \cdot u[y]$ for `z = x * y`, `z = x .* y`, and `z = cross(x, y)`.
- Division[3], $u[z] = u[x]/u[y]$ for `z = x / y`, and `z = x ./ y`.
- Trigonometic function, $u[z] = 1, u[x] = \text{rad}$ for `z = sin(x)`, `z = cos(x)`, and `z = tan(x)`.
- Inverse trigonometic function, $u[z] = \text{rad}, u[x] = 1$ for `z = asin(x)`, `z = acos(x)`, and `z = atan(x)`.
- Differentation, $u[z] = u[x]/\text{s}$ for `z=der(x)`.
- Dimensionless function, $u[z] = 1$, $u[x] = 1$ for `z = log(x)`, `z = log10(x)`, `z = exp(x)`, `z = sinh(x)`, `z = cosh(x)`, and `z = tanh(x)`.
- Propagate first argument, $u[z] = u[x]$ for `z = -x`, `z = skew(x)`, `z = fill(x, ...)`, `z=hold(x)`, `z = noEvent(x)`, `z = noClock(x)`, `z = min(x)`, `z = max(x)`, `z = abs(x)`, `z = sample(x, ...)` (clocked version), `z = superSample(x, ...)`, `z = subSample(x, ...)`, `z=backSample(x, ...)`, `z = shiftSample(x, ...)`, `z = previous(x)`, `z = scalar(x)`, `z = vector(x)`, `z = matrix(x)`, `z = promote(x)`, `z = transpose(x)`, `z = diagonal(x)`, `z = delay(x, ...)`, `z = pre(x)`, `z = actualStream(x)`[2], and `z = inStream(x)`[2].

[2] Implemented after testing

[3] With special treatment of literals

- Smooth, $u[z] = u[y]$, for `z = smooth(p, y)`.
- Reinit, $u[x] = u[y]$ for `reinit(x, y)`.
- Root, $u[z]^n = u[x]$ for `z = nthRoot(x, n)` (if we can evaluate n) and with $n = 2$ for `z = sqrt(x)`.
- Relations $u[x] = u[y]$ for `z = x > y`, `z = x >= y`, `z = x < y`, `z = x <= y`, `z = x == y`, and `z = x != y`.
- Time-units, $u[z] =$ s, for `z=time`, and `z=interval()`
- Truncated division $u[z] = 1$, $u[x] = u[y]$ for `z = div(x, y)`[2].
- Rounding $u[z] = 1$, $u[x] = 1$ for `z = floor(x)`[2], `z = ceil(x)`[2], and `z = integer(x)`[2].
- Arctan2 $u[z] =$ rad, $u[x] = u[y]$ for `z=atan2(x, y)`[2].
- Semilinear $u[z] = u[m] \cdot u[x]$, $u[x] = u[y]$ for `z = semiLinear(m , x, y)`[2].
- Integer power, $u[z] = u[x]^n$ for `z=x^n` if integer expression n (and ignored unless we can evaluate n).
- Non-integer power, $u[z] = 1$, $u[x] = 1$, $u[y] = 1$ for `z=x^y` if y is not an integer expression.
- Sample, $u[x] =$ s, $u[y] =$ s for `sample(x, y)`[2] (non-clocked version).
- Condition, $u[z] = u[x]$, $u[y] = u[x]$ for `z=`**`if`**` c `**`then`**` x `**`else`**` y` if `c` is not an evaluated expression.
- Only propagate arrays if either side has homogenous unit, $u[z] = u[x_1] = \ldots = u[x_n]$ for `z = {`x_1`, ..., `x_n`}`, `z = [`x_1`, ..., `x_n`]`, `z=[`x_1`; ...; `x_n`]`, and `z = cat(p, `x_1`, ..., `x_n`)` (ignoring `p`).
- Array subscripting `z = x[i]` if `i` is literal then treat $u[x_i]$ as scalar, otherwise only propagate from homogenous $u[x]$ to $u[z]$,
- Special case for literals and `zeros(...)` (the latter to handle **`equation`** `a.f + b.f = zeros(3);`).
- Connect-statements are handled by checking the equations generated from the connection-sets, the details will only impact diagnostics for incorrect models.

B Gcd-case

For completeness we will present the gcd-case for the algorithm, even if we haven't seen any practical models requiring this.

```
model GcdCase
  Real x;
  parameter Real y=1;
  Real area(unit="m2");
  parameter Real T(unit="s");
equation
  der(area)=x^4*y^6;
  T*area=x^4*y^(-6);
end GcdCase;
```

After some substitutions we get the unit-equations:

$$1 = u[x]^4 \cdot u[y]^6 \cdot \frac{s}{m^2} \tag{33}$$

$$1 = u[x]^4 \cdot u[y]^{-6} \cdot \frac{1}{s \cdot m^2} \tag{34}$$

for the first equation we introduce a new variable

$$u[z] \to u[x] \cdot u[y] \tag{35}$$

giving (36)

$$1 = u[z]^4 \cdot u[y]^2 \cdot \frac{s}{m^2} \tag{37}$$

$$1 = u[z]^4 \cdot u[y]^{-10} \cdot \frac{1}{s \cdot m^2} \tag{38}$$

importantly we solve the *same* equation again for y

$$u[y] \to (m^2 \cdot s)^{1/2} \cdot u[z]^{-2} \tag{39}$$

$$1 = u[z]^{24} \cdot \frac{s^4}{m^{12}} \tag{40}$$

giving

$$u[z] \to \frac{s^{1/6}}{m^{1/2}} \tag{41}$$

$$u[y] \to s^{-1/6} \tag{42}$$

$$u[x] \to m^{1/2} \tag{43}$$

The example is as contrived as it looks, but it shows that the case can be handled. The introduced variable has exponents corresponding to truncated integer division for all of the other exponents, which corresponds to one step in the normal algorithm for greatest common divisor. It stops when the new smallest exponent divides all others, and that exponent is the greatest common divisor (in this case 2).

DOI 10.3384/ecp21817