Extendable Physical Unit Checking with Understandable Error Reporting

Peter Aronsson[1] David Broman[2]

[1]MathCore Engineering AB, Teknikringen 1F
SE-58330 Linköping, Sweden, peter.aronsson@mathcore.com
[2]Dept. of Computer & Information Science, Linköping University, davbr@ida.liu.se

Abstract

Dimensional analysis and physical unit checking are important tools for helping users to detect and correct mistakes in dynamic mathematical models. To make tools useful in a broad range of domains, it is important to also support other units than the SI standard. For instance, such units are common in biochemical or financial modeling. Furthermore, if two or more units turn out be in conflict after checking, it is vital that the reported unit information is given in an understandable format for the user, e.g., "N.m" should preferably be shown instead of "m2.kg.s-2", even if they represent the same unit. Presently, there is no standardized solution to handle these problems for Modelica models. The contribution presented in this paper is twofold. Firstly, we propose an extension to the Modelica language that makes it possible for a library designer to define both new base units and derived units within Modelica models and packets. Today this information is implicitly defined in the specification. Secondly, we describe and analyze a solution to the problem of presenting units to users in a more convenient way, based on an algorithm using Mixed Integer Programming (MIP). Both solutions are implemented, tested, and illustrated with several examples.

Keywords: dimensional analysis, unit checking, dimensions, error reporting, language design

1 Introduction

Modelica is a full fledged object-oriented equation-based modeling language. However, its expressiveness can sometimes lead to models containing errors that are hard to detect and isolate[3].

One important area where modeling errors can give devastating consequences is inconsistency of physical units and dimensions within equations and component connections. We have earlier proposed a design and made a prototype implementation for dimensional inference and unit consistency checking[1] in the MathModelica[7] and OpenModelica[9] tools. Such checking will help the users by reporting at compile time if they have made a unit inconsistency error in their model. However, this becomes less useful when modeling something that cannot be expressed in the dimensions as defined by the SI standard. This is a common scenario for biochemical modeling based on the SBML standard[12]. Such models frequently use the non-standard dimension "Item" for counting, e.g., molecules. MathModelica has a translator tool[2] for translating SBML models into Modelica (and vice versa). To make the translation tool more robust, user defined units should be considered in the dimensional analysis too. Another example is in financial applications, where it is required to use the dimension "money".

It is also a problem that the system of units (and potential extensions) is not described in the Modelica language standard, i.e., the language specification only specifies how to parse unit expressions, not what the units mean, or how the checking should be performed. This may result in that tools from different vendors are not compatible, where some tools accept certain Modelica libraries, while others reject them due to unit inconsistency.

Another problem is how to present the resulting units (e.g. from unit inference) to the user, when one or more units are inconsistent. For instance, presenting the unit "m2.kg.s-2A-1" to the user is not very understandable. Instead, the tool should translate this into a more appropriate derived unit (or combination of derived units and base units), like "Wb" or perhaps "V.s". The preferred choice of these two might be different depending on domain and context. For instance, if this unit is reported in a domain of Magnetic models, "Wb" might be preferred, but if it is reported in a context where only units "V", "A" and

"Ohm" are used it is probably more appropriate to use "V.s". The presentation should not only contain standard SI units, but also extended units defined by the user and these could also be selected by regarding domain and context information.

In this paper we propose a solution to these problems by specifying both base units and derived units in a generic way, so that new dimensions easily can be added. We propose this as an extension to the Modelica language so that different Modelica tools can behave alike. At the same time, the library developer is also given a more powerful mechanism for specifying nonstandard units in a uniform way. Section 2 presents the proposed Modelica language extension that enables the model user to describe both base units and derived units. In Section 3 we show a new method of how a tool can interpret the dimensional units inferred by the type checker and presents unit errors to the user in a more readable form. This is done by formulating a mixed integer programming (MIP) problem that will select more appropriate units depending on both context and potentially also user preferences. We have made an implementation and an evaluation in the MathModelica and OpenModelica tools, described in Section 4. Finally, Section 5 contains related work and Section 6 concludes the paper.

2 Extendable Unit Definitions

The Modelica specification [10] includes a section describing the syntax of unit expressions, i.e., how for example an expression such as `"kg.m/s2"` should be parsed. However, besides a reference to ISO standard 31/0-1992, no information is given regarding the *semantics* of how to perform the actual unit checking. This general openness of the specification makes it possible for different tool vendors to implement their own way of handling unit checking, giving implementation freedom, but also limits the possibility for models to be exchanged and treated in the same way by different tools.

Instead of letting a reference to an ISO standard define the meaning of base units (e.g. "V" and "s") and derived units (e.g. "N.m"), we propose in this section that the definition should be stated directly in the source code of Modelica classes. Possible benefits with this approach are:

- Tools from different tool vendors use and interpret exactly the same set of unit definitions.
- Besides the standard SI units, it is easy for users and library developers to add both new base and derived units for a particular library.

Our goal is that both this work with extendable unit definition and our previous work on general unit checking should form a foundation for a new semantic description of units in the Modelica specification. Even though we today have a running test implementation, the work is still at an early stage, and more work on formalizing the semantics is required for inclusion in the specification. Moreover, our intension is not that unit checking should be a core part of the specification. Instead, we propose that such a language feature should be defined as an *optional module* in the specification, enabling tool vendors to explicitly choose and officially state if the functionality of such a module is supported.

2.1 Requirements

We have during the design work of extendable unit definitions for Modelica considered the following requirements:

- *Backwards compatibility*. Models designed with the earlier definitions where the meaning of units was implicit, should also work in a new environment where the units are defined by the library developer.
- *Only library definitions.* Both base units and derived units should be able to be added by library developers, i.e., the tools should not have any prior knowledge about defined units.
- *Declarative and easy to use*. The new extension for defining new units must be declarative in the sense that the order of definitions should not matter. It must also be easy to use, e.g., defined units should be stated in a user friendly format such as "N.m"; not using its unit vector format.
- *Weights for different domains.* It should be possible to prioritize certain units for a specific domain, to enable better error reporting.
- *Prefixes are pre-defined*. Prefixes, such as "m" for milli and "k" for kilo are pre-defined in the specification, i.e., these are not extendable.

Following these requirements, an overview of our design proposal is outlined in the following three subsections.

2.2 Informal Syntax

Adding new syntax to a language is the least interesting and challenging issue from a language design point of view, but results nevertheless often in large

debates at design meetings. Hence, the following proposed syntax is only for presentation purpose and can most likely be changed in a version that is accepted for inclusion in the Modelica specification.

We introduce a new keyword **`defineunit`**, which is used both for defining new base units and derived units. For example, to define the three first base units of the SI-system, the following lines can be added to an arbitrary Modelica class.

```
defineunit m;
defineunit kg;
defineunit s;
```

Derived units are defined by combining base units or other derived units. For example, to define the derived unit Newton, the following line is added.

```
defineunit N(exp="m.kg.s-2");
```

The expression consists here only of base units. The syntax of the unit expression is the same as the syntax specified in the current Modelica standard. However, it would be very inconvenient if the derived units always must be defined using base units. Hence, we allow expressions also to include other derived units. For example, this line would define the derived unit Pascal:

```
defineunit Pa(exp="N/m2");
```

Note that both a derived unit (N) and a base unit (m) are used in the unit expression.

There is also an optional parameter weight that can be used for specifying how important an unit is in the domain. This is used by the algorithm presented in Section 3 for better error reporting. If no weight argument is specified, a default value of 1 is used. The weight can also be specified explicitly by using a named parameter. For example

```
defineunit Pa(exp="N/m2", weight=2);
```

states that Pascal is a unit that is more important in this library and will therefore have higher priority when used in error reporting.

2.3 Formal Syntax

The **`defineunit`** extension can be defined in the EBNF grammar of the Modelica specification, by adding the following production:

```
unit_clause :
  defineunit IDENT
      [ "(" named_arguments ")" ]
```

The `unit_clause` is then used inside the element production as follows:

```
element :
  unit_clause |
  ...
```

Where `...` mean the rest of the right side of the original `element` production.

2.4 Informal Semantics Overview

The semantics of the extendable unit definition is not intended to be described in detail here. Instead, the intent is to give a brief overview of how a compiler can treat the unit definitions. A more complete and formalized description is postponed as future work in conjunction with a language extension proposal for the Modelica language design group.

From the syntax description, it is clear that unit definitions can be placed anywhere in the element section of a class. Hence, units can be defined within any restricted class, e.g., packages, models, and functions. When checking equations and/or statements within a model, two passes are performed. In the first pass, all components and sub-components of the model are traversed and unit definitions collected. This includes searching both the components' scope and their parents' scope. In the second pass, the ordinary instantiation/elaboration takes place. During this elaboration, equations and statements are checked for unit consistency using the unit definitions collected in the first pass.

The order of how the unit definitions are collected in the first pass is not important. If the set of unit definitions contains several elements with the same unit name, it is an error if their respective unit expressions are different. For example, if Newton (N) is defined more than once, each definition must have the same expression, i.e., `"m.kg.s-2"`. After elimination of identical unit definitions, the resulting set of unit definitions is used to generate an internal normalized representation of units. Following the approach outlined in our previous work [1], each unit is then represented in a vector format. To be able to generate this normalized form, it is required that all definitions and dependences between derived units and base units form a directed acyclic graph (DAG). Hence, derived units are not allowed to be defined so that they form cyclic structures. If such a cyclic structure is detected, an error should be reported. For example, the following definitions should be rejected:

```
defineunit U1(exp="m.U2");
defineunit U2(exp="U3/s");
defineunit U3(exp="U1.kg");
```

If several unit definitions exist with the same name and expression, but with different weights, these weights are used later in pass two for better error reporting. The weights for unit definitions with the same unit name are multiplied together, forming the new weight. For example, if the following definitions of Pascal exist:

```
defineunit Pa(exp="N/m2", weight=1.5);
defineunit Pa(exp="N/m2", weight=2);
```

The resulting unit definition is:

```
defineunit Pa(exp="N/m2", weight=3);
```

In the current implementation a library must redefine all types that should be treated with a different weight factor. For example, if a library would like to have higher weights on Pascal, types that are using `Pa`, such as `Pressure`, must be redefined in the library. The main rationale for this design choice is better performance of the instantiation/elaboration process of the compiler.

3 Reporting Units

The unit checker described in previous work[1] uses a vector of seven rational numbers; one for each dimension. The reason for using rational numbers is to be able to handle a `sqrt` function or exponents of arbitrary rational numbers, e.g., x^(2/3), which is very commonly used in engineering equations. In this work, the length of this vector is determined by the number of dimensions added to the system. The library developer adds all definitions of base units and derived units to the standard library, including the standard SI units (see Section 2). Every unit is thus described by a vector of at least 7 elements. For instance, the unit Watt ("W") corresponding to the base units "m2.kg.s-3" is described by the vector (2,1,-3,0,0,0,0). The problem is, given a sought unit with dimension vector $\dim_t$ (the target unit), to find a linear combination of units (both derived and base) that matches the dimension vector $\dim_t$. But, in order to select more appropriate units we should prefer units that are close to the target unit. Also, we should prefer to use derived units instead of base units, as this will probably be closer to what an engineer expects.

As a first attempt, we can formulate the problem as:

For a target unit, t, that has dimensional vector $\dim_t$

minimize

$$\sum_{j=1}^{NU} p_j x_j \quad \text{where } p_j = \frac{1}{w_j}(1+|\dim(j)-\dim_t|)$$

$$\text{subject to } \sum_{j=1}^{NU}\dim(j)x_j = \dim_t$$

Where

- NU is the number of units (base and derived)
- w_j is a real number > 0
- dim(j) is the dimensional vector for the j:th unit
- x_j is the sought exponent for each unit
- |v| is the L2 norm of vector v

This formulation works fine as long as x_j is a positive integer value. If negative values were allowed those would contribute negatively to the objective, and thus favor negative exponents over positive ones. So, to allow negative exponents in units we must handle them separately. This can be done by instead setting up the problem as:

$$\text{minimize} \sum_{j=1}^{2NU} p_j x_j$$

where

$$p_j = \frac{1}{w_j}(1+|\dim(j)-\dim_t|)$$

$$w_{j+NU} = w_j \qquad ,1 < j < NU$$

$$\dim(j+NU) = -\dim(j) \qquad ,1 < j < NU$$

$$\text{subject to } \sum_{j=1}^{2NU}\dim(j)x_j = \dim_t$$

With the formulation above we double the problem size and represent negative exponents with a set of separate variables. The weights for the newly introduced variables are identical to its positive correspondent exponent, and the dimensional vector is negated.

If the dimensional units only were described with Integers (e.g. as done in Dymola v.7 [5]), this formulation would be sufficient. However, because we allow Rational numbers as exponents and because it is most likely that derived units should be expressed only by integers, we need to reformulate the problem. We let the variables of base units be of type Real (or preferably rational) and the derived units be of type integer, thus leading to a mixed integer programming problem.

3.1 Example

Let us consider an example. For simplicity we limit the example to use three base units (m,s,kg) and define four derived units according to Table 1 below.

Unit	*Vector representation*
m	(1,0,0)
kg	(0,1,0)
s	(0,0,1)
N	(1,1,-2)
Pa	(-1,1,-2)
J	(2,1,-2)
W	(2,1,-3)

Table 1. A subset of the SI units.

Suppose that a unit of a certain term is inferred to "m.kg2.s-3", corresponding to the vector representation (1,2,-3).. If we use (1,1,1,1,1,1,1) as weight vector the problem becomes:

$$\text{minimize } p\,x$$

$$\text{subject to } m\,x = \dim_t$$

$$m = \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 2 & 2 & -1 & 0 & 0 & -1 & 1 & -2 & -2 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & -1 & 0 & -1 & -1 & -1 & -1 \\ 0 & 0 & 1 & -2 & -2 & -2 & -3 & 0 & 0 & -1 & 2 & 2 & 2 & 3 \end{pmatrix}$$

$$p = 1 + \begin{pmatrix} \sqrt{13} & \sqrt{11} & \sqrt{21} & \sqrt{1} & \sqrt{6} & \sqrt{3} & \sqrt{2} & \sqrt{17} & \sqrt{19} & 3 & \sqrt{38} & \sqrt{34} & \sqrt{43} & 3\sqrt{6} \end{pmatrix}$$

$$\dim_t = (1,2,-3)$$

The m matrix sets up the constraints for the dimensions, the first seven columns corresponds to the values in Table 1 above, and the seven last columns are their negated values. The criteria vector *p* gives the weight for each variable as the distance of the dimension vector of that dimension to the target dimension plus one. The reason for adding one to the distance is to be able to control that even selecting a perfect match can be avoided by using weights. For instance, the first element has a value of $1+\sqrt{13}$ since the distance (norm) from (1,0,0) to (1,2,-3) plus one is

$1+\sqrt{13}$ $(1+\sqrt{(1-1)^2+(0-2)^2+(0--3)^2}$).

When solving this problem it will give the values: (0,1,0,1,0,0,0,0,0,1,0,0,0,0)

which correspond to the unit "kg.s-1.N".

By adjusting the weight vector different results are obtained. For instance, if we increase the weight only for "Pa" the results instead become:

(0,0,1,0,1,1,0,0,0,0,0,0,0,0)

which correspond to unit "s.Pa.J", i.e. it prefers to use unit "Pa" in the result.

3.2 Use of Rational Numbers

So far we have not used any rational numbers in our examples. So how does rational numbers affect the proposed solution?

Since we formulated the problem as a MIP (Mixed Integer Programming), it can allow both integer variables and real variables. The idea is to limit the derived units to integer values, so that units like "W-(1/3)" are not produced. Otherwise it will be hard for the user to find out what is missing to correct the error, since the user himself has to translate the derived units into base units and then apply the exponent.

As an example we will take the unit "W(1/2)", which corresponds to the unit vector (1,1/2,-3/2,0,0,0,0) The solution when derived units are integers and base units are reals becomes: "kg-(1/2).s(1/2).N". If the problem is solved with all variables as real values[1] the solution is instead: "N(1/2).Gy(1/4)" which is much harder for a user to interpret.

An alternative formulation could be to instead formulate the linear programming problem using only integers, by multiplying the base unit vector by the greatest common divisor among the rational numbers, and then solve the corresponding integer linear programming (ILP) problem. The solution must then be divided by the greatest common divisor. The problem with this formulation is that it can not guarantee that derived units are only expressed in integer exponents. For example, given the unit vector (5/2,3/2,-9/2,0,0,0,0), the corresponding MIP solution becomes "m-1/2.kg-1/2.s-1/2.N.J". However, transforming to ILP gives "Pa.J.W.Gy", which results in "Pa(1/2).J(1/2).W(1/2).Gy(1/2)", which is hard for a user to understand.

3.3 Adjusting Weights According to Context

As illustrated by the example above, the weights for each unit can be modified to control the solution. This can be used to guide the solver into selecting units that are preferred for a given context. For instance, let us consider a simple equation for calculating the power:

$$P = RI^2$$

[1] The real values are "Rationalized" before presentation by approximation to rational numbers with small integers.

Suppose the variables P and I have defined units of "W" and "A" respectively. The resistance is inferred to "m2.kg.s-2.A-2" elsewhere (i.e. missing a s-1 to be "Ohm"). If this problem is solved it will regardless of weights result in "Wb", since that will result in a perfect match, giving the lowest cost (since the distance is zero, the cost will be $1/w_j$). However, a user might be more familiar if units closer to "W" and "A" is used. By adjusting the weights (increasing "W" and "A", and decreasing the weight of the rest of the derived units so they are smaller than weights of base units), the result instead becomes: "s-1.A-2.W". Of course, if it instead is evident from the context that Ohm is the preferred unit, we could decrease its weight and increase the rest of the derived units, resulting in: "s-1.Ohm".

In conclusion, the resulting unit can be controlled by modifying the weights of derived units. To find out the weights one could look at the current context the unit is defined in. For instance, in an electrical component that does not have any units from the magnetic domain declared, the weights of the units "Wb","T" and "H" could be decreased.

The possibility we have chosen is to let the library developers themselves define the weights according to their preferred units. This is the suggested approach described in Section 2.

3.4 Minimizing the Number of Used Derived Units

One problem with the proposed solution is that the same minimal value can be obtained by either selecting a mixture of several derived units or by selecting multiples of only one derived unit. For example, let us consider the unit vector for "Ohm3", which corresponds to the vector (6,3,-9,-6,0,0,0). With ones as weight, the result becomes "F-1.Ohm.H"; this is not preferable. If weights of units are adjusted according to previous section this might be avoided, but it is not always the case that a context of units may help (the context may be empty).

An alternative is to make an automated adjustment on the units to try to minimize the use of derived units. This can be expressed by the following algorithm:

1. Run the MIP problem with standard weights (or user preferred weights).
2. If several derived units are reported, increase weight on one of them and rerun MIP problem. If less derived units are reported, keep the adjusted weight and repeat 2, otherwise try next derived unit. Repeat until all derived units reported has been tried.

Let us try this idea on our example. As stated above the first run of the problem gave "F-1.Ohm.H" as result. We first increase the weight of "F" and rerun. This gives "Ohm3" as we expect. Same result is also given if we increase weight for "H". However, if we increase the weight of "Ohm", the result becomes "F-1.S-1.H", which is clearly not a good choice.

4 Implementation and Evaluation

A prototype for reporting units has been implemented in Mathematica and a full implementation is now completed in the MathModelica/OpenModelica frontend.

4.1 Testing the Modelica Standard Library

The unit checking and error reporting functionality have been tested in MathModelica on the Modelica Standard library v 2.2.1, which is the latest version where unit checking corrections (based on Dymola version 7.0 unit checking functionality) of the library have not been performed. The unit checker reported the same problems as Dymola did on version 2.2.1 and after applying the corrections made in version 2.2.2, the affected models passed the unit check. This gives an indication of that both tools behave correctly, or at least they behave in the same manner.

However, there are some cases in the standard library where Dymola does not report unit errors, but MathModelica and OpenModelica does. One such case is the use of the built-in `exp` function, which is used in e.g. Semiconductor models in the Electrical package. The problem can be illustrated with this simple model:

```
model UnitProblem
 Real i(unit="A");
 Real v(unit="V") = 2.4;
equation
 i = exp(v);
end UnitProblem;
```

Dymola does not report any errors for this model, even though the exp function should have a dimensionless argument and give a dimensionless return value. Thus, since the MSL is primarily developed with Dymola, the unit conversion corrections that are done for other models are not done for models containing `exp`, `log`, and other dimensionless built-in

functions. This is also reflected in the `Modelica.Math` library where these functions are declared as unspecified unit with e.g., `input Real x;` instead of dimensionless, using input

```
Real x(unit="1");
```

To correct these defects we propose to make the exponent function, logarithm function, and others, that are dimensionless to be declared with unit "1" in the MSL, and that the usage of these functions in the library are corrected so a dimensionless unit is passed and returned from these functions.

4.2 Unit Extendibility

The unit extendibility has been tested and evaluated by adding unit definitions (`defineunit`) for all SI base units and derived units according to [5]. These definitions have been added to `SIunits.mo` in the Modelica standard library. Preliminary tests show that this approach is backwards compatible compared to having these definitions built-in, i.e., unit checking works as expected even if the SI units are defined in the standard library. Models have also been tested, where additional base units (e.g. a currency base unit of "USD") were added.

4.3 Usability of Error Reporting

Preliminary tests have been conducted for evaluating the usability and readability of errors when different weights are used in different libraries. However, further more comprehensive tests must be performed in the future to verify that the reported units are indeed understandable.

5 Related Work

Unit checking exists in several Modelica tools, such as Dymola[9] and Simulation X[6]. There are also unit checking and dimensional analysis in other non Modelica related tools and languages. See the "Future work" section in previous paper [1] for these references.

To our knowledge, no earlier work has been published on how to select units for presentation. Tools with unit checking have for certain some way of selecting which units to present to the user but the method of how this is done is not clearly stated, and the user can not affect the outcome as is suggested in this paper.

6 Conclusions

We have showed a new method of solving the problem of presenting inferred and inconsistent units by the unit checker in a format that is more understandable for the user. The method is based on forming a mixed integer programming (MIP) problem to decide which base units and derived units to use in the communication with the user. We have also proposed an extension to the Modelica language, where unit definitions can be stated within any restricted class, making it possible to define new user defined units that are not part of the standard SI units.

A prototype has been implemented in Mathematica, followed by a complete implementation in MathModelica and OpenModelica. The same unit errors on the Modelica standard library that Dymola have detected were also reported by our tool, but we also detected more inconsistent units, and proposed further corrections of the standard library.

7 Acknowledgements

This research was funded by CUGS (National Graduate School in Computer Science), MathCore Engineering, the Swedish Research Council (VR), and the Biobridge project supported by the European Commission in the sixth framework programme.

References

[1] D. Broman, P. Aronsson, P. Fritzson, "Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica", 6Th International Modelica Conference, March 3-4, 2008, Bielefeld, Germany.

[2] J. Brugård et. Al, "Creating a Bridge between Modelica and the Systems Biology Community", 7th International Modelica Conference, Como, Italy, 2009.

[3] Peter Bunus, "Debugging Techniques for Equation-Based Languages". Ph.D. Thesis. Department of Computer and Information Science. Linköping University. 2004.

[4] Bureau international des poids et mesures (BIPM). Le Système international d'unités, The International System of Units. Organisation intergouvermentale de la Convention du Mètre, 8th Edition.

[5] Dynasim. Dymola version 7.0 http://www.dynasim.com [Last access: August 23, 2009].

[6] ITI. SimulationX. http://www.iti.de/ [Last access: August 20, 2009].

[7] MathCore. MathModelica http://www.mathcore.com [Last access: August 23, 2009].

[8] Mathematica. Wolfram Research Inc. http://www.wolfram.com. [Last access: August 23, 2009]

[9] S.-E. Mattson, H. Elmqvist, "Unit Checking and Quantity Conservation", 6Th International Modelica Conference, March 3-4, 2008, Bielefeld, Germany.

[10] Modelica Association. "Modelica - A Unified Object-Oriented Language for Physical Systems Modeling Language Specification Version 3.1" 2009. Available from http://www.modelica.org.

[11] The OpenModelica Project. Available from: http://www.openmodelica.org

[12] Systems Biology Markup Language (SBML), Available from: http://www.sbml.org