M. Sandberg, B. Lisper:
Dimensional Analysis for Modelica.
Modelica Workshop 2000 Proceedings, pp. 129-136.

Paper presented at the Modelica Workshop 2000, Oct. 23.-24., 2000, Lund, Sweden.

All papers of this workshop can be downloaded from
http://www.Modelica.org/modelica2000/proceedings.html

Dimensional Analysis for Modelica

Mikael Sandberg
Mälardalen University,
mikael.sandberg@mdh.se

Björn Lisper
Mälardalen University
bjorn.lisper@mdh.se

Abstract

Dimensional analysis (DA) checks for dimensional inconsistencies in equations and expressions. It provides the ability to make fast estimates of the dimensional correctness of equations in Modelica. We believe this will enhance the overall quality of Modelica code. Incorporating a DA system into Modelica can be done without any changes to the language itself. However, we believe that a slightly extended type definition is needed to incorporate dimensional analysis in a practical yet effective manner for Modelica.

1. Introduction

Dimensional Analysis (DA) is a static analysis that we believe will increase the quality of the Modelica code (equations in particular). It accomplishes this by doing dimensional consistency checks, common to scientific and engineering computations.

DA should not be mixed up with unit checking where knowledge on the different units and the conversion between them are necessary. The difference is often lost in everyday use of the two terms. Therefore we think a definition of the terms is called for:

A *unit* is a scale factor, unit factor and a dimension. The combination of the two factors and the dimension yields a *unit*.

A *dimension* is a physical quantity.

An example of this is length. Length is a dimension but not a unit. The unit could be meter, foot, inch or centimeter, which all have the dimension of length.

DA made its way into computer languages in the late 70´s [Kar78] but has not made a big impact. We believe this to be the opposite for Modelica since the program domain of Modelica is mainly dimensional, e.g., a Modelica model is part of one or more physical domains that inherently is dimensional.

Besides Modelica's intentions to unify the specification of multi domain models it is constructed to ease the transition from theory to computer simulation. This is actually one of the most important properties for the users and creators of Modelica models. The amount of expressions and equations in Modelica programs and models increases with the complexity of the system that is being simulated, and the risk of human error increases accordingly. DA will help to automate the transition and therefore contribute to the overall quality of the Modelica models.

A DA system can be thought of as an extension to the type systems of programming languages. It extends the types with another property; dimension. DA is a step towards a full unit-check system, where faults like the NASA Mars orbiter thrust malfunction [Isb99] would have been detected as early as in the compilation phase of the project.

The article presents a combination of two known constructs: dimensional analysis and type inference. This is applied on an explicitly typed language in general and an idea on how this might be adopted for Modelica is given.

We show this system in an example and discuss the need and possibilities to incorporate a DA system into Modelica.

2. Related Work

DA has been constructed for numerous languages: [Bar95, Umr94] accomplishes DA in C++ by using templates. They manage to do this without the need to extend the language. Before templates where part of the ANSI C++ standard DA was done with data abstractions [Cme88]. Hilfinger [Hil88], uses the inherent abstraction facilities of Ada to present a DA system for Ada in the form of a package. This package was also done without any extension to the language. Furthermore, work has been done for languages such as ML [Ken96] where the language type system was used to enable dimension inference.

The representations of dimensions have also been subject to change over the years, towards a more

symbolic and flexible manner ([Hil88] vs. [Ken96]). Still the selection of representation for dimensions is mainly based on the originating language rather than on convenience.

Type inference [Car87, Hal96] has been presented for many programming languages, mainly functional languages like Haskell and ML.

3. Dimensional Analysis

Based on the SI system of measurements a dimensional space that represents the domain of all SI units is created. This vector space is spanned by seven unit vectors representing the dimensions in the SI system: Time, Mass Length, Electric charge, Thermodynamic temperature, Amount of substance and Luminous intensity.

There are two ways of describing this vector space: numerically and symbolically.

The numerical description directly uses seven-dimensional vectors (or 7-tuples) where the elements corresponds to the exponent of that particular base dimension, for example:

$$\mathcal{D}_{Time} = \langle 1,0,0,0,0,0,0 \rangle$$
$$\mathcal{D}_{Length} = \langle 0,0,1,0,0,0,0 \rangle$$

These dimensions can be combined to describe other physical quantities like:

$$\mathcal{D}_{Velocity} = \langle -1,0,1,0,0,0,0 \rangle$$

This tuple description was used by Barton and Nackman [Bar95] for C++ (see Section 2).

The symbolic representation uses a textual variant of the same dimensional vector, where *dimension symbols* represent the dimensions:

$$\mathcal{D}_{Time} = Time$$
$$\mathcal{D}_{Length} = Length$$

These dimensions, like the tuple description, can be combined to describe other physical quantities like:

$$\mathrm{D}_{Velocity} = Time^{-1}, Length$$

Formally, the symbolic representation is an abelian group with a binary operation ",". It is easy to see that for a fixed set of dimension symbols, the symbolic representation is isomorphic with the tuple representation. An advantage with the symbolic representation is that it easily can be extended to represent dimensions outside the selected system of measurement (in our case SI). For example: bits per second are not a valid SI unit since it is not a derivative of any base dimension. However, it is easy to introduce a new dimension symbol *Bit*. Bits per second can now be represented symbolically as follows:

$$\mathcal{D}_{bps} = \mathcal{D}_{Bit/Time} = Bit, Time^{-1}$$

This is not possible using the tuple representation since the seven base dimensions are fixed.

Dimensions can naturally be calculated with dimensional operators that corresponds to normal arithmetic operators of multiplication, and division for the tuple representation:

$$\langle t_1,...,t_n \rangle \hat{*} \langle t'_1,...,t'_n \rangle = \langle t_1 + t'_1,...,t_n + t'_n \rangle$$

$$\langle t_1,...,t_n \rangle \hat{/} \langle t'_1,...,t'_n \rangle = \langle t_1 - t'_1,...,t_n - t'_n \rangle$$

Equivalent operators exist for the symbolic representation (see [Ken96]). For simplicity, we will in the following use the tuple representation.

As a short example, consider the well known equation of velocity:

$$velocity = length / time$$

Using the definition of time, length, velocity and the dimensional operators the dimensional correctness could be checked in the following manner using the tuple representation:

$$\mathcal{D}_{Velocity} = \mathcal{D}_{Length} \hat{/} \mathcal{D}_{Time} \Rightarrow$$
$$\langle -1,0,1,0,0,0,0 \rangle = \langle 0,0,1,0,0,0,0 \rangle \hat{/} \langle 1,0,0,0,0,0,0 \rangle \Rightarrow$$
$$\langle -1,0,1,0,0,0,0 \rangle = \langle -1,0,1,0,0,0,0 \rangle \Rightarrow True$$

When all variables and constants of a program are assigned dimensions (as in the example above), a dimensional check can be performed to show the dimensional correctness of that program; this might not always be the case. Some of the program constructs may lack dimensional information and then the DA system must be extended with an inference system. This inference system will try to infer dimensions on the constructs that are missing dimensional information. This is formalized as follows.

We use a small language to exemplify this system. It contains expressions and equations only. Here, *e* stands for expressions, "id" for identifiers, "num" for numerical constants, and *f* for functions.

$$e \rightarrow e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2$$
$$\mid idnum \mid f(e_1,...,e_n)$$
$$idnum \rightarrow \text{id} \mid \text{num}$$
$$equation \rightarrow e_1 = e_2$$

The system also needs a function *dim* that assigns a dimension to each identifier and numerical constant:

$$dim(idnum): \begin{cases} \propto_{idnum} \text{ if not dimensionalized} \\ \mathcal{D}_e \text{ if dimensionalized} \end{cases}$$

Here, $\propto$ denotes an unbound dimension, that is: a unique variable that will be assigned a dimension by the inference system. For example if *x* is an dimensionally unbound identifier, using the tuple representation, then *dim*(*x*) is a 7-tuple of unique variables:

$$dim(x) = \propto_x = \langle x_1,...,x_7 \rangle$$

The inference system tries to assign a dimensional type to each expression *e*. Formally, it has typing judgements of the following form:

$$e : \mathcal{D}, E$$

Here, $\mathcal{D}$ is the dimension of the expression *e* (could be unbound) and *E* a set of dimensional constraints (equations) that have been derived from *e*.

The following axiom assigns a dimension type to identifiers and numerical constants:

$$idnum : dim(idnum), \varnothing$$

Note that the dimension of *idnum* could be unbound, e.g., does not have any dimension, and that this will later be bound by the inference system.

These are the inference rules for dimensional typing of arithmetical operations over expressions:

$$\frac{e_1 : \mathcal{D}_1, E_1 \qquad e_2 : \mathcal{D}_2, E_2}{e_1 + e_2 : \mathcal{D}_1, E_1 \cup E_2 \cup \{\mathcal{D}_1 = \mathcal{D}_2\}}$$

$$\frac{e_1 : \mathcal{D}_1, E_1 \qquad e_2 : \mathcal{D}_2, E_2}{e_1 - e_2 : \mathcal{D}_1, E_1 \cup E_2 \cup \{\mathcal{D}_1 = \mathcal{D}_2\}}$$

$$\frac{e_1 : \mathcal{D}_1, E_1 \quad e_2 : \mathcal{D}_2, E_2}{e_1 * e_2 : \mathcal{D}_1 \hat{*} \mathcal{D}_2, E_1 \cup E_2}$$

$$\frac{e_1 : \mathcal{D}_1, E_1 \quad e_2 : \mathcal{D}_2, E_2}{e_1 / e_2 : \mathcal{D}_1 \hat{/} \mathcal{D}_2, E_1 \cup E_2}$$

This is the inference rule for equations:

$$\frac{e_1 : \mathcal{D}_1, E_1 \qquad e_2 : \mathcal{D}_2, E_2}{e_1 = e_2 : \mathcal{D}_1, E_1 \cup E_2 \cup \{\mathcal{D}_1 = \mathcal{D}_2\}}$$

This is the inference rule for general function application:

$$\frac{e_1 : \mathcal{D}_1, E_1 \ldots e_n : \mathcal{D}_n, E_n \quad e : \mathcal{D}_e, E_e}{f(e_1,...,e_n) : \mathcal{D}_e, E_e \cup E_1 \cup \ldots \cup E_n}$$

Here, *e* is the expression defining the return value of the function.

Finally, here are inference rules for some selected functions in Modelica:

$$\frac{e : \mathcal{D}_e, E_e}{der(e) : \mathcal{D}_e \hat{/} \mathcal{D}_{Time}, E_e}$$

der(e) returns the derivative of e with the respect to time.

$$\frac{e : \mathcal{D}, E}{sin(e) : \propto_e, E_e}$$

The dimension of the *sin* function is an unbound dimensional type.

A set S of expressions, or equations, yields a set of dimensional equations $\{E(e) \mid e \in S\}$ through the inference system. This set corresponds to a linear equation system. There are three different cases:

1. No solution: dimensions are not correct.
2. Unique solution: dimensions are correct.
3. Parametric solution: dimensions undeterminable, but can be given a minimal, parameterized description as a linear system of equations.

4. A Dimensional Inference System for Modelica

To show a working DA system for Modelica no change to the syntax of the language is necessary. Extending the Modelica language construct of *unit* suffices. According to the semantic specification of Modelica the *Unit* part of the type declaration complies with ISO 31/0-1992 “General principles concerning quantities, units and symbols” and ISO 1000-1992 “SI units and recommendations for the use of their multiples and of certain other units”. They do not depict a formal syntax and therefore Modelica can permit universal strings as units.

Using *unit* as a string permits the storage of dimensional data for types in Modelica.

The *unit* part of type declarations is optional in Modelica. Therefore, this mechanism can yield partially dimensonalized programs. Dimensional inference can still be able to derive dimensional properties for such programs. We exemplify this with Modelica functions using the tuple representation of dimensions:

```
type Volt   = Real(Unit="<-2,1,2,-1,0,0,0>");
type Ampere = Real(Unit="<-1,0,0,1,0,0,0>");

function Resistance
   input Volt u;
   input Ampere i;
   output Real r;

algorithm
   r := u/i;
end Resistance;
```

Without the dimensional inference system the parameter *r* does not have a declared dimension and consequently the function Resistance does not have a dimensional return value. The dimension of *r* can however, be inferred by the fact that both *i* and *u* are dimensionalized (e.g., have a dimension declared or inferred) together with the fact that the operator / is defined to combine two dimensions forming a new (see inference rules). This in the conjunctions with the assignment statement results in the fact that *r* must have the same dimension as *u/i* namely resistance.

Another example is when it is not possible to infer a dimension:

```
type Volt   = Real(Unit="<-2,1,2,-1,0,0,0>");

function Capacitance
   input Volt u;
   input Real i;
   output Real c
algorithm
   c := i/der(u);
end Capacitance;
```

The dimensional information available is insufficient to allow any inferred dimension for either *c* or *i*. However, their possible dimensions will be linked through a linear equation.

The trivial case, not shown here, is when all the constructs have dimensional information and the problem is reduced to a dimensional check.

5. Adapting Modelica for Dimension Inference

As mentioned above extending the *unit* statement of the type declaration to incorporate DA into Modelica is preferable. This adoption should be extended for the possibility of a full unit-check system right a way. This implies that not only dimensional information has to be stored within the type but also information on the scale factor and unit factor is needed. This combined with the existing textual description of the type declaration would be sufficient to handle DA and unit checking as well as presenting a textual representation of predefined dimensions and units to users.

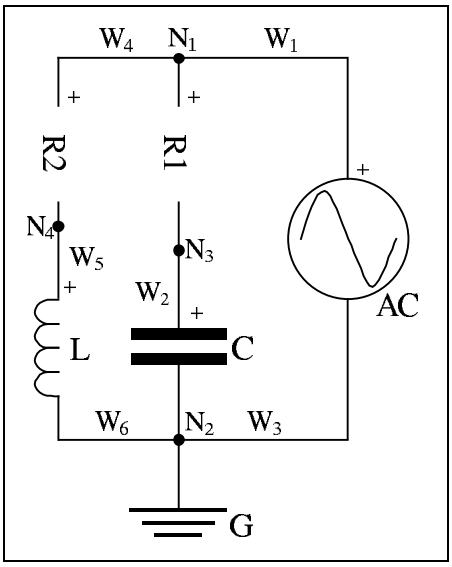
We suggest the a syntactical extension of Modelica to support statements as follows example:

```
type Volt=Real(unit="V", dimension="<-2,1,2,-
1,0,0,0>", sfactor=1,ufactor=1);
```

6. An Example

The example circuit is taken from the introduction of Modelica [Fri98] and modified to reflect a more current version of the electric library of Modelica. This example does not use the suggested dimensional extension to the Modelica language and for reasons of clarity uses the tuple representation of dimensions.

Fig 1. Simple electric circuit



Modelica Code

The following declarations of types is included to reflect dimensions:

```
type Volt = Real(unit="<-2,1,2,-1,0,0,0>");
type Ohm = Real(unit="<-1,1,2,-2,0,0,0>");
type Ampere = Real(unit="<-1,0,0,1,0,0,0>");
type Capacitance =
Real(unit="<2,0,0,1,0,0,0>");
type Inductance = Real(unit="<0,1,2,-
2,0,0,0>");
```

Here is the actual model of the circuit:

```
class circuit
      Capacitor C(C=0.01);
      Inductor L(L=0.1);
      Ground G;
      Resistor R1(R=10);
      Resistor R2(R=100);
      VsourceAC AC;

   equation
      connect(AC.p, R1.p);
      connect(R1.n, C.p);
      connect(C.n, AC.n);
      connect(R1.p, R2.p);
      connect(R2.p, L.p);
      connect(L.n, C.n);
      connect(AC.n, G.p);
end circuit;
```

To further show the equations behind this example a part of the Modelica electrical library is included:

```
connector Cut
   Volt across[:] = zeros(0);
   flow Ampere through[:] = zeros(0);
   Boolean booleans[:] = fill(false, 0);
end Cut;

connector CutBase extends Cut;
end CutBase;

connector PinCurrentIn extends CutBase;
end PinCurrentIn;

connector PinCurrentOut extends CutBase;
end PinCurrentOut;

connector TwoPinNeg extends PinCurrentOut
end TwoPinNeg;

connector TwoPinPos extends PinCurrentIn;
end TwoPinPos;

model TwoPin
   Volt   v;
   Volt   Vp;
   Ampere i;
   Volt   Vn;

   TwoPinPos p(across=[Vp], through=[i]);
   TwoPinNeg n(across=[Vn], through=[-i]);

equation
   v  = Vp - Vn;
end TwoPin;

model Resistor extends TwoPin;
   parameter Ohm R;

equation
   R*i = Vp - Vn;
end Resistor;

model Capacitor extends TwoPin;
   parameter Capacitance C;

equation
   C*der(v) = i;
end Capacitor;

model Inductor extends TwoPin;
   parameter Inductance L;

equation
   L*der(i) = Vp - Vn;
end Inductor;

model Ground
   Volt Vp;
   Ampere i;

equation
   Vp = 0;
end Ground;

model VsourceAC extends TwoPin;
   constant Real PI = 3.141592653589793;
   parameter Volt VA = 220;
   parameter Real freq = 50;
   parameter Real t0 = 0;

equation
   Vp-Vn = VA*sin(2*PI*freq*(Time-t0));
end VsourceAC;
```

Now selecting a few of the equations from this example (all would be to extensive for this article) as S, the set of equations:

$$S=\left\{\begin{array}{l} R*i \;=\; Vp\;\; Vn, \\ L*der(i) \;=\; Vp \text{ - } Vn, \\ C*der(v) \;=\; I, \\ Vp\text{-}Vn \;=\; VA*\sin\left(2*PI*freq*(Time\text{-}t0)\right) \end{array}\right\}$$

For clarity the scope of the variables are not shown here. *R* should be *Resistor.R* and *L* should be *Inductor.L* etc. The dimensional vector space is also reduced to a four degrees to reflect the electric sub-space (domain) of the SI system of measurements. This is done mainly to shorten the amount of text presented in the derivations and has no effect on the calculations performed therein.

When run through the inference system the equations of S gives a system of dimensioned equations. These equations can then be checked for integrity.

$$\frac{\dfrac{R:\langle -1,1,2,-2\rangle,\varnothing \qquad i:\langle -1,0,0,1\rangle,\varnothing}{R*i:\langle -1,1,2,-2\rangle \hat{*} \langle -1,0,0,1\rangle,\varnothing} \qquad \dfrac{Vp:\langle -2,1,2,-1\rangle,\varnothing \qquad Vn:\langle -2,1,2,-1\rangle,\varnothing}{Vn-Vp:\langle -2,1,2,-1\rangle,\{\langle -2,1,2,-1\rangle=\langle -2,1,2,-1\rangle\}}}{R*i=Vp-Vn:\langle -2,1,2,-1\rangle,\{\langle -2,1,2,-1\rangle=\langle -2,1,2,-1\rangle,\langle -2,1,2,-1\rangle=\langle -2,1,2,-1\rangle\}}$$

Derivation of the dimensional equation for $R*i=Vp-Vn$

$$\dfrac{\dfrac{L:\langle 0,1,2,-2\rangle,\varnothing \quad der(i):\langle -1,0,0,1\rangle \hat{/} \langle 1,0,0,0\rangle,\varnothing}{L*der(i):\langle 0,1,2,-2\rangle \hat{*} \langle -2,0,0,1\rangle,\varnothing} \quad \dfrac{Vp:\langle -2,1,2,-1\rangle,\varnothing \quad Vn:\langle -2,1,2,-1\rangle,\varnothing}{Vn-Vp:\langle -2,1,2,-1\rangle,\{\langle -2,1,2,-1\rangle = \langle -2,1,2,-1\rangle\}}}{L*der(i)=Vp\text{-}Vn:\langle -2,1,2,-1\rangle,\{\langle -2,1,2,-1\rangle = \langle -2,1,2,-1\rangle,\langle -2,1,2,-1\rangle = \langle -2,1,2,-1\rangle\}}$$

Derivation of the dimensional equation for $L*der(i)=Vp\text{-}Vn$

$$\dfrac{\dfrac{C:\langle 2,0,0,1\rangle:\varnothing \quad der(v):\langle -2,1,2,-1\rangle \hat{/} \langle 1,0,0,0\rangle:\varnothing}{C*der(v):\langle 2,-1,-2,2\rangle \hat{*} \langle -3,1,2,-1\rangle:\varnothing} \quad i:\langle -1,0,0,1\rangle:\varnothing}{C*der(v)=i:\langle -1,0,0,1\rangle,\{\langle -1,0,0,1\rangle = \langle -1,0,0,1\rangle\}}$$

Derivation of the dimensional equation for $C*der(v)=i$

$$\dfrac{\dfrac{Vp:\langle -2,1,2,-1\rangle,\varnothing \quad Vn:\langle -2,1,2,-1\rangle,\varnothing}{Vn-Vp:\langle -2,1,2,-1\rangle,\{\langle -2,1,2,-1\rangle = \langle -2,1,2,-1\rangle\}} \quad \dfrac{VA:\langle -2,1,2,-1\rangle:\varnothing \quad \dfrac{2:\propto_4,\varnothing \quad \dfrac{PI:\propto_3,\varnothing \quad \dfrac{freq:\propto_2,\varnothing \quad \dfrac{Time:\langle 1,0,0,0\rangle,\varnothing \quad t0:\propto_1,\varnothing}{Time\text{-}t0,\langle 1,0,0,0\rangle,\{\langle 1,0,0,0\rangle = \propto_1\}}}{freq*(Time\text{-}t0):\propto_2 \hat{*} \langle 1,0,0,0\rangle,\{\langle 1,0,0,0\rangle = \propto_1\}}}{PI*freq*(Time\text{-}t0):\propto_3 \hat{*} \propto_2 \hat{*} \langle 1,0,0,0\rangle,\{\langle 1,0,0,0\rangle = \propto_1\}}}{2*PI*freq*(Time\text{-}t0):\propto_4 \hat{*} \propto_3 \hat{*} \propto_2 \hat{*} \langle 1,0,0,0\rangle,\{\langle 1,0,0,0\rangle = \propto_1\}}}{\sin(2*PI*freq*(Time\text{-}t0)):\propto_5,\{\langle 1,0,0,0\rangle = \propto_1\}}}{VA*\sin(2*PI*freq*(Time\text{-}t0)):\langle -2,1,2,-1\rangle \hat{*} \propto_5,\{\langle 1,0,0,0\rangle = \propto_1\}}}{Vp\text{-}Vn = VA*\sin(2*PI*freq*(Time\text{-}t0)):\langle -2,1,2,-1\rangle,\{\langle 1,0,0,0\rangle = \propto_1,\langle -2,1,2,-1\rangle = \langle -2,1,2,-1\rangle \hat{*} \propto_5\}}$$

Derivation of the dimensional equation for $Vp\text{-}Vn = VA*\sin(2*PI*freq*(Time\text{-}t0))$

7. Conclusion and Further Research

As stated in the introduction a DA system is a structured start at a full unit check system for programming languages. Dimensions are the fundamental base on which unit checking is transformed to a linear problem. Two different units with the same dimensions are but a linear transformation from conversion.

The examples and the DA system shown in this article use the SI metric unit system, but the DA system can be adapted to any set of dimensions and units.

We have shown that a dimensional inference system behaves as a type inference system and that this can be applied to an explicitly and strongly typed language such as Modelica.

We believe that the symbolic representation is more flexible and conforms more to the multi-domain requirements of Modelica. We therefore recommend its use over the tuple representation.

We will present a formal proposal to the extension of the type declaration to handle DA and unit checking to the Modelica design group.

An implementation of the DA system for Modelica, most likely by extending an existing implementation, is the next logical step in the evolution

8. Acknowledgments

We wish to thank Jan Gustafsson and Jan Carlson for their insightful discussions on the subject.

References

[Bar95] J. Barton, L. Nackman, Dimensional Analysis, C++ Report, January 1995, p39-43

[Car87] L. Cardelli, Basic Polymorphic typechecking, 1987, Science of Computer Programming 8, Elsevier Science Publishers B.V, Pages 147-172

[Cme88] R. F. Cmelik, N. H. Gehani, Dimensional Analysis with C++, 1988, IEEE Software May 88, Pages 21-27

[Fri98] P. Fritzson, V. Engelson, Modelica - a Unified Object Oriented Language for System Modeling and Simulation. European. Conference on Object-Oriented Programming, 1998, Brussels

[Hal96] C. V. Hall et al, Type Classes in Haskel, 1996, ACM Transactions on Programming Languages and Systems, Vol. 18, No 2, March 1996, Pages 109-138

[Hil88] P. N. Hilfinger, An Ada Package for Dimensional Analysis, ACM Transactions on Programming Languages and Systems, Vol. 10, No. 2, April 1988, Pages 189-203.

[Isb99] D. Isbell et al, Mars Climate Orbiter Team Finds Likely Cause of Loss, 1999, http://mars.jpl.nasa.gov/msp98/news/mco990930.html

[Kar78] M. Karr, D. B. Loveman III, Incorporation of units into programming languages, 1978, Communications of the ACM May 78, Pages 385-391

[Ken94] A. Kennedy, Dimension Types, European Symposium on Programming, 1994, LNCS Volume 788

[Ken96] A. Kennedy, Type Inference and Equation Theories, 1996, Technical Report Laboratoire D'Informatique Ecole Polytechnique

[Mod99] Modelica – A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification version 1.3, 1999, Modelica Design Group

[Umr94] Z. Umrigar, Fully Static Dimensional Analysis with C++, 1994, ACM SIGPLAN Notices, Volume 29, No 9, Pages 135-139