MASTER'S THESIS | LUND UNIVERSITY 2016

Units of Measurement in a Modelica Compiler

Daniel Eliasson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-32

Units of Measurement in a Modelica Compiler

Daniel Eliasson
daniel.c.eliasson@gmail.com

August 21, 2016

Master's thesis work carried out at Modelon AB.

Supervisor: Niklas Fors, niklas.fors@cs.lth.se
Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Modelica is a declarative object-oriented language for describing mathematical models. Modelica models are used for simulation and optimization of systems and processes. The models are described using variables and equations. It is possible to associate units of measurement to variables, for example, variable x is in m/s. This thesis presents a way to analyze units in equations to find errors. This approach has been implemented in the compiler JModelica.org.

Keywords: dimensional analysis, units, Modelica

Acknowledgements

I would like to thank Jesper Mattsson and Jonathan Kämpe, my external supervisors at Modelon.

Contents

Chapter 1
Introduction

When designing physical systems, such as cars and powerplants, it is very useful to be able to test new ideas rapidly. Traditionally prototyping has been used for this purpose, but for a while now this practice has been complemented with computer simulations. Simulation and modeling software can help test ideas before a prototype has even been made, and can help to validate the solution and minimize the risk of surprising behaviour. Simulations can also be used to optimize processes.

Modelica[4] is a declarative object-oriented language for describing mathematical models. Modelica models of physical systems and processes are used for simulation and optimization to better understand the behavior of a real system. The models are described using variables and equations. It is possible to associate units to variables. This thesis presents a way to analyze units in equations, to find errors in Modelica models. We have implemented our approch in JModelica.org[9], an open-source development environment for Modelica.

1.1 Problem Description

Units of measurement are similar to a type system, but for physical quantites[1]. Values with different units may be of different kinds, for example a length and a velocity. Some operations are non-sensical when applied to different kinds of values. What does it mean to add a length and a velocity? Unit checking is about finding the places where units mismatch, similar to how type checking is done in a statically typed language.

The Modelica Language Specification[12] defines a way to specify physical units for variables, for example variable *x* is in *m/s*. However it does not define any semantics for units. Without any defined semantics for units, tools are free to completely ignore units on variables. Some tools try to check that units in equations and expressions match. This thesis presents a solution for checking units, with an implementation in JModelica.org.

In the example below variables `a` and `b` have different units. The equation declares

that a and b are equal. But a and b do not have matching units so the equation doesn't make sense.

```
model M
  Real(unit="m") a;
  Real(unit="s") b;
equation
        a = b;
end M;
```

One way to correct the problem in the previous example, is to use a constant, as illustrated below. How these errors are to be corrected depends on the situation. But in order to correct the errors, we first need to be able to find them. That is what this thesis is about.

```
model M
  Real(unit="m") a;
  Real(unit="s") b;

  constant Real(unit="m/s") c = 1;
equation
        a = c*b;
end M;
```

There is a lot of questions to answer. How should units be represented internally in the compiler in a way that is easy to check for equality and to print in an error message? How should the compiler deal with variables without a unit? Is there a way to derive the unit of a variable without a defined unit and use that to do further checking?

The Modelica Language Specification also specifies a way to associate another unit with each variable called a display unit. The display unit should be used to display the value of the variable in simulation results and user interfaces. To do that the value of the variable need to be converted to the display unit. A model is compiled into a functional mock-up unit (FMU). An FMU is a tool independent format for model exchange, defined in the FMI standard [3]. Some information about units is included in the FMU, for example conversions between units and display units. As we are already creating a framework for units for the purposes of unit checking it is convenient to use the same framework for conversions as well.

In the example below the variable v is in meters per second during simulation but the result should be converted to kilometers per hour before displayed to the user.

```
Real(unit="m/s", displayUnit="km/h") v;
```

1.2 Results

We have made an implementation that can check units in equations, where all variables and constants are declared with units. When we tested our solution on the Modelica Standard Library we found 2 errors in the library.

1.3 Related Work

Unit checking has already been added to other Modelica tools. Unit checking in Dymola is similar to our own in that it is conservative [11]. Dymola's solution also need all units and constants to have units to be able to find errors. In OpenModelica there is an implementation of unit checking that can infer the unit of variables that have no declared unit [5]. Our internal representation of units is similar to the one in OpenModelica. Unit checking has also been implemented for imperative object oriented languages (eg. Java, C#, etc) [1].

Chapter 2
Background

In this chapter we go into more depth about Modelica, JModelica.org, and dimensional analysis.

2.1 Modelica

Modelica is a non-proprietary, object-oriented, declarative language for mathematical modeling. The language and the Modelica Standard Library (MSL) has been developed by the Modelica Association since 1996 [4]. Modelica models are used to model physical systems. Different tools can then use the models to simulate and optimize the systems. JModelica.org [9], OpenModelica[7] and Dymola[13] are examples of some of the tools available. Modelica models are built using variables and equations. Models can also be built by connecting other models together which allows for encapsulation and reuse.

Listing 2.1 is an example model for simulating the trajectory of a canonball. For simplicity we have left out the distance traveled and air resistance. We only keep track of the height above the ground. The relationships between acceleration and velocity, and between velocity and height, are described using differential equations. The function `der` is a built-in operator for time derivative. The `parameter` keyword is used to mark variables that are constant during simulation, but that can be changed between simulations. The result of the simulation can be seen in figure 2.1.

In listing 2.1, all the variables and parameters have the type `Real`. The type `Real` is used to represent a real number. It also has some extra attributes. Examples of attributes are minimum value, maximum value, starting value, unit and display unit. If not specified, the value of the unit attribute is the empty string, meaning that no unit is declared for the variable. Because no units are declared in the example, we are missing out on the possibility of checking if the units match. In a small example like this one it would be easy to spot any errors. For larger and larger systems there is increased value in automatically detecting errors, like adding a length to a velocity or an equation with acceleration on one

side and force on the other.

Listing 2.1: Modelica model describing a canonball without units

```
model Canonball
  parameter Real h0 = 10   "Initial height above ground";
  parameter Real v0 = 15   "Initial vertical velocity";
  parameter Real g  =-9.82 "Gravitational acceleration";

  Real h "Height above ground";
  Real v "Vertical velocity";
initial equation
  h = h0;
  v = v0;
equation
  g = der(v);
  v = der(h);
end Canonball;
```

In listing 2.2 the code defines subtypes of `Real` with different units. The new types are then used when defining parameters and variables. It is not nessecary to define these types over and over again, as there are definitions for many different kinds of quantities in the standard library (package Modelica.SIunits). In this example, all the variables have a defined unit, so units can be checked automatically. As an additional benefit, the names of the types serve as documentation for the variables, as they name what kind of quantity they hold. Note though that both models give exactly the same simulation result (see figure 2.1). The units do not affect the calculations made during the simulation.

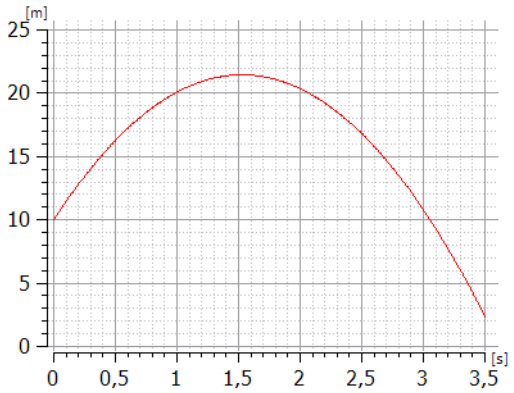
Listing 2.2: Modelica model describing a canonball with units

```
model Canonball
  type Length       = Real(unit = "m");
  type Velocity     = Real(unit = "m/s");
  type Acceleration = Real(unit = "m/s2");

  parameter Length       h0 = 10   "Initial height above ground";
  parameter Velocity     v0 = 15   "Initial vertical velocity";
  parameter Acceleration g  =-9.82 "Gravitational acceleration";

  Length   h "Height above ground";
  Velocity v "Vertical velocity";
initial equation
  h = h0;
  v = v0;
equation
  g = der(v);
  v = der(h);
end Canonball;
```

Figure 2.1: Simulation result of the cannonball example.



2.2 JModelica.org

JModelica.org is an Open source platform for simulation and optimization using Modelica[10]. It was created as a result of research at the Department of Automatic Control at Lund University [9]. It is currently maintained by Modelon AB. The Modelica compiler, part of JModelica.org, is implemented using the JastAdd metacompiler[8]. We have implemented unit checking and unit conversions in the compiler.

2.3 Dimensional Analysis

The analysis of the relationships between quantites and their physical dimensions is called *Dimensional Analysis*[1]. Examples of physical dimensions are length, time, mass, force, area, current, etc. Quantities are amounts in those dimensions. For example the speed of light is a quantity, in this case an amount of speed.

Units are known quantites used to define other quantites. For example, $3\,\mathrm{m}$ is a quantity that is 3 times the length of the unit meter (abbreviated m). Units can be combined to form new units according to certain rules. Below are some rules that apply to units of variables and expressions. In the examples below, variables X and Y are some arbitrary variables, and unit is the function that maps variables and expressions to units.

$$\begin{aligned}
\mathrm{unit}(X \cdot Y) &= \mathrm{unit}(X) \cdot \mathrm{unit}(Y) \\
\mathrm{unit}(X \,/\, Y) &= \mathrm{unit}(X) \,/\, \mathrm{unit}(Y) \\
\mathrm{unit}(X + Y) &= \mathrm{unit}(X) = \mathrm{unit}(Y) \\
\mathrm{unit}(X - Y) &= \mathrm{unit}(X) = \mathrm{unit}(Y)
\end{aligned}$$

Below are some rules that apply directly to units. Where u, v and w are some arbitrary units, and $\mathbf{1}$ is the dimensionless unit used for scalefactors and ratios.

$$
\begin{aligned}
u \cdot \mathbf{1} &= u \\
u \cdot v &= v \cdot u \\
u \cdot (v \cdot w) &= (u \cdot v) \cdot w \\
u^0 &= \mathbf{1} \\
u^1 &= u \\
u^n \cdot u &= u^{n+1} \\
u^n / u &= u^{n-1}
\end{aligned}
$$

Here are some examples where we can see some of these rules in practise:

$$
\begin{aligned}
3\,\mathrm{m} \,/\, 6\,\mathrm{s} &= 3 \,/\, 6\,\mathrm{m\,s^{-1}} = 0.5\,\mathrm{m\,s^{-1}} \\
12\,\mathrm{kg} \cdot 3\,\mathrm{m\,s^{-2}} &= 12 \cdot 3\,\mathrm{kg\,m\,s^{-2}} = 36\,\mathrm{kg\,m\,s^{-2}}
\end{aligned}
$$

All units and quantities exist in a physical dimension, such as length, mass, time. Dimensions combine in a similar way to units. It is possible to express a quantity in different units as long as the units are in the same dimension, for example length. This is called unit conversion. Note that the quantity stays unchanged. It is only expressed in a different way. The scale factor applied to the numbers when converting are called the conversion factor. For example:

$$3\,\mathrm{m} = 9.842\,519\,69\,\mathrm{feet}$$

The International System of Units (SI) defines 7 base dimensions with 7 corresponding base units [6, p. 105, p. 116]. From these, a set of coherent SI units are derived [6, p. 117]. Coherent SI units are all combinations of unscaled SI units, for exmaple, m, $\mathrm{m\,s^{-1}}$, $\mathrm{m^2}$, $\mathrm{kg\,m\,s^{-1}}$. Among these are 22 named units (eg. $\mathrm{N} = \mathrm{kg\,m\,s^{-1}}$). All named units also have their own symbol. It also has a system of prefixes that create an array of units with different scales from each coherent SI unit [6, p. 121].

The dimensions of a quantity or unit can be always be expressed as a product of powers of the seven base dimensions as follows: [6, p. 105]

$$\dim(X) = \mathrm{L}^a \cdot \mathrm{T}^b \cdot \mathrm{M}^c \cdot \Theta^d \cdot \mathrm{I}^e \cdot \mathrm{N}^f \cdot \mathrm{J}^g$$

The base dimensions are length L, time T, mass M, thermodynamic temperature Θ, electric current I, amount of substance N, and luminous intensity J. We will later use this property to represent units.

Chapter 3
Implementation

This chapter is about the particulars of our implementation of units in JModelica.org.

3.1 Unit Syntax

When a unit is associated with a variable in Modelica, the unit is described using a string, for example, “kg.m/s2”. We will refer to these strings as *unit expressions* in the text below. The Modelica Specification defines a syntax for unit expressions [12, p. 245]. The grammar is quite short, having only 9 productions. Below is an abbreviated version, the full grammar can be found in Appendix A.

```
unit_expression:
    unit_numerator [ "/" unit_denominator ]
unit_numerator:
    "1" | unit_factors | "(" unit_expression ")"
unit_denominator:
    unit_factor | "(" unit_expression ")"
unit_factors:
    unit_factor [ "." unit_factors ]
unit_factor:
    unit_operand [ unit_exponent ]
```

Because the grammar was rather simple we decided to write a recursive descent parser for it. In hindsight, a more maintainable solution would be to use a lexer- and parser-generator instead. When using a generator you write a description that is much shorter and easier to read than handwritten code. The grammar has 9 productions, and with a little bit of code for each production, the resulting description would be perhaps 30 to 40 lines of code. Our parser is 256 lines of code.

The parser transforms unit expression strings into an internal representation. In the process, the parser also checks that the expression is syntactically correct. There is no semantic checking done, because all syntactially valid units are considered valid. The result of parsing a unit expression is an object representing the unit.

3.2 Representation

We use an internal representation that is easier to work with than unit expression strings. What makes the internal representation more suitable depends on what we want to do. The question is then, what operations do we need?

We need to be able to compare two units. This is used when unit checking to check if two units match. It is difficult to do this directly using the unit expressions, because a single unit can be described using multiple different strings. For example, "m/s", "m.s-1" and "s-1.m" all represent the same unit. It would be good to use some normalized form so that all objects representing the same unit have the same internal representation.

When calculating the unit of expressions, we need to be able to combine units to form new units. How units of expressions are calculated is described in 3.3 Analysis.

We also need to be able to generate a string from a unit for use in error messages. It is important that the unit is recognizable to users in error messages. Some units can be written in different ways, for example, $\mathrm{J} = \mathrm{kg\,m^2\,s^{-2}} = \mathrm{N\,m} = \mathrm{Pa\,m^3}$. Different expressions are used in different context, for example, J is used for energy while $\mathrm{N\,m}$ is used for work and torque.

It should also be possible to generate the conversion factor and offset between two units. This information is needed in the simulation binary generated by the compiler.

Variables and expressions with unknown units need special treatment. We do not want to report errors when there is not enough information to know if it's really an error. To handle this a special class is used for this case (null object pattern).

The way we have choosen to represent units, can be seen in listing 3.1. Each unit is comprised of an array of seven integers for the dimension, a float for the scale, a float for the offset and a string for storing the original string expression.

Listing 3.1: The internal representation in java code

```
class Unit {

    private final double scale;
    private final double offset;
    private final int[]  powers;
    private final String originalExp;

    ...
}
```

The `scale` is used to to represent the relative scale of different units, for example $1\,\mathrm{km}$ is 1000 times larger than $1\,\mathrm{m}$, therefore km has a scale of 1000 and m a scale of 1. The scale is relative to the unscaled SI units (also known as coherent SI units). The `offset` field is only used for temperature scales such as °C (degrees Celsius) where the zero point

is offset from SI unit K (Kelvin). The `powers` array is used to encode the dimension of the unit. In Section 2.3 Dimensional Analysis, we mentioned that all dimensions can be expressed as a product of the powers of the seven base dimensions. The powers in that equation (repeated below) are saved in the `powers` array (in order a through g). Finally, the original unit expression is stored in `originalExp`. It is used in error messages.

$$\dim(X) = \mathrm{L}^a \cdot \mathrm{T}^b \cdot \mathrm{M}^c \cdot \Theta^d \cdot \mathrm{I}^e \cdot \mathrm{N}^f \cdot \mathrm{J}^g$$

Below are some examples of using this representation. All examples will use the following notation:

```
(scale, offset, [powers], name)
```

First a couple of simple examples.

```
(1.0, 0.0, [1, 0, 0, 0, 0, 0, 0], "m")
(1.0, 0.0, [0, 1, 0, 0, 0, 0, 0], "s")
(1.0, 0.0, [1,-1, 0, 0, 0, 0, 0], "m/s")
(1.0, 0.0, [2, 0, 0, 0, 0, 0, 0], "m2")
```

The next example illustrates the use of prefixes. Here we can see from the scale that $1\,\mathrm{km}$ is $1000\,\mathrm{m}$ and that $1\,\mathrm{km}^2$ is $1\,000\,000\,\mathrm{m}^2$.

```
(1000.0,    0.0, [1, 0, 0, 0, 0, 0, 0], "km")
(1000000.0, 0.0, [2, 0, 0, 0, 0, 0, 0], "km2")
```

For historical reasons, the kilogram is the only base unit that contains a prefix. Multiples of the kilogram are constructed by applying prefixes to the gram. For this reason the gram has a scale of 0.001 so that the scale of the kilogram becomes 1.0.

```
(0.001, 0.0, [0, 0, 1, 0, 0, 0, 0], "g")
(1.0,   0.0, [0, 0, 1, 0, 0, 0, 0], "kg")
```

Some units have aliases. In the example below we can see that “N”, “m.kg/s2” and “m.s-2.kg” are the same unit because they have the same scale, offset and powers. The original unit expression is saved as a string and can be used in error messages. Similarly “V” is “m2.s-3.kg.A-1”.

```
(1.0, 0.0, [1, -2, 1, 0, 0, 0, 0], "N")
(1.0, 0.0, [1, -2, 1, 0, 0, 0, 0], "m.kg/s2")
(1.0, 0.0, [1, -2, 1, 0, 0, 0, 0], "m.s-2.kg")

(1.0, 0.0, [2, -3, 1, 0, -1, 0, 0], "V")
(1.0, 0.0, [2, -3, 1, 0, -1, 0, 0], "m2.s-3.kg.A-1")
```

Currently Modelica only supports SI units, but non-SI units are supported by our internal representation. The following illustrates how a non-SI unit would be represented. In the example the imperial unit foot is used. From the scale factor we can see that one foot is 0.3048 meters.

```
(0.3048, 0.0, [1, 0, 0, 0, 0, 0, 0], "ft")
```

3.3 Analysis

In order to check units in equations, we first need to be able to calculate the unit of expressions. Let us take a look at one of the examples from before.

Listing 3.2: ”Example model of a cannonball with units.”

```
model Canonball
  type Length       = Real(unit = "m");
  type Velocity     = Real(unit = "m/s");
  type Acceleration = Real(unit = "m/s2");

  parameter Length       h0 = 10   "Initial height above ground";
  parameter Velocity     v0 = 15   "Initial vertical velocity";
  parameter Acceleration g  =-9.82 "Gravitational acceleration";

  Length   h "Height above ground";
  Velocity v "Vertical velocity";
initial equation
  h = h0;
  v = v0;
equation
  g = der(v);
  v = der(h);
end Canonball;
```

Let us take a closer look at one of the equations: `v = der(h)`. To calculate the unit of the left hand side, we first need to look up `v` and then get its unit. To calculate the unit of the right hand side, we look up and get the unit of `h`, then we use that to calculate the unit of `der(h)`. An overview of the process can be seen in Figure 3.1, where the abstract syntax tree for the equation is annotated with the information flow.

In general the process can be divided into three steps. The first step is to look up the unit of variables. To do this we use the existing system for name lookup to find the variable declarations. Then we find and parse the unit attribute of each variable into internal representation.

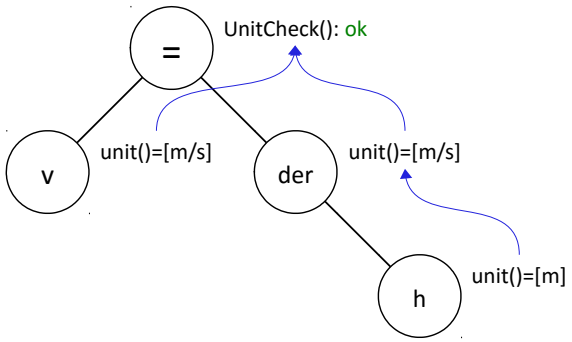


Figure 3.1: Information flow for unit checking of equation `v = der(h);`

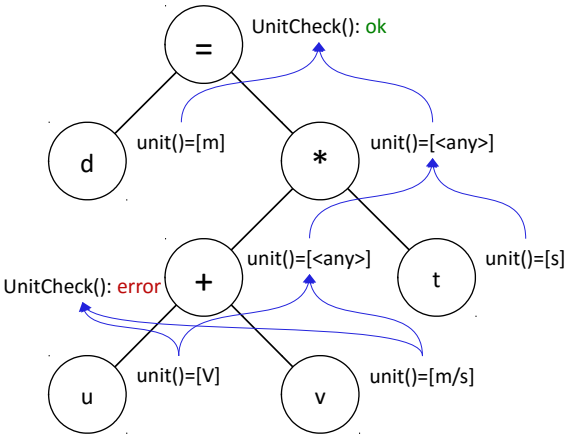


Figure 3.2

The next step is calculating the unit of expressions by combining the units of their respective subexpressions. In 2.3 Dimensional Analysis, we went through some of the rules for how it works for different operators.

Finally, when the unit of each of the expressions on either side of an equation is known, we can check to see if they match. If they do not match, then generate an error message, else all is fine, do nothing.

In the next example we go into some of the details of this process. In particular what happens when a unit error is present. The equation below has four vairables, d is a length in meters, v is a velocity in metes per second, u is a voltage in volts, and t is a duration in seconds.

$$d = (v + u) \cdot t$$

The units don't match when adding v and u together, generating an error. To avoid follow up errors the unit of the addition expression get the special value `AnyUnit` . `AnyUnit` matches any other unit and is propagated when calculating the unit of expressions. See figure 3.2.

Here is an error printout from the example:

```
Error at line 16, column 13, in file 'MyModel.mo':
   Units in expression doesn't match.
Left expression: u          Unit: V
Right expression: v         Unit: m/s
```

Chapter 4
Evaluation

To evaluate our solution, we have tested our solution on example models from Modelica Standard Library. Unit tests have been used to check individual aspects of our solution.

4.1 Correctness

To check that our solution is correct, we have a set of unit tests. We have 7 tests, each in 2 different versions. One positive version without errors and one negative version with unit errors. The positive tests are there just in case errors are generated when then should not be, ie. when all unit match. The result of positive tests is usually indistinguishable from not having unit checking at all, which is why the negative test is needed as well. The negative tests check that errors are generated when they should be. We also have another 4 tests with only one version. These tests check that various warnings are generated.

To further test our solution we tested it on the Modelica Standard Library (MSL). MSL contains around 350 example models, that use the models defined in the library. The example models are already used to test JModelica.org. We tested our solution on these examples. If our solution finds any unit errors that would either mean that there are errors in MSL or that there are errors in our implementation.

First time we ran the tests from MSL, we found a couple of errors that should not have been reported as errors. We decided to make changes such that these were reported as warnings instead. We ran the tests again and found that 7 out of 366 models had unit errors. We later traced these back to 2 distinct errors in the Modelica Standard Library (one of the errors caused multiple tests to fail).

We also tested our solution on tests from OpenModelica. OpenModelica has unit checking, but also unit inference which our solution does not have. Many of the tests from OpenModelica that test unit checking, also test the unit inference. We have modified the tests to remove the need for unit inference. Our solution passes all the tests. Out of curiosity we run our unit tests in OpenModelica and it passed all but one test. We check

if `unit` and `displayUnit` are units in the same dimension, while OpenModelica does not. For example, it is wrong to use `m` as `unit` and `s` as `displayUnit` because `m` is a Length unit and `s` is a Time unit.

Chapter 5

Future work

It would be useful to be able to infer the unit of a variable that does not have a declared unit. At first it may seem strange, why would some variable not have any declared unit? The answer is generic models, small pieces that can be reused in different contexts, which means the unit is not known in advance. There are examples of this in MSL. For example the gain model (it is actually a block, we are simplifying here). A gain can be used whenever a variable needs to be multiplied with a value. Below is an example model M that uses this. The gain model is similar, but a simplified version, of the one in MSL (Modelica.Blocks.Math.Gain).

```
model gain

  parameter Real k(start=1, unit="1");

  input Real u "Input signal";
  output Real y "Output signal";

equation

  y = k*u;

end Gain;
```

```
model M
  import SI = Modelica.SIunits;

  SI.Voltage u;
  SI.Voltage v;

  Gain g(k=1.5);

equation

  u = 1;
  g.u = u;
  v = g.y;

end M;
```

With inference the compiler could detect that both `g.u` and `g.y` are voltages, and check that `u` and `v` have the same unit. Without inference, as is the case in our solution, this goes unchecked. In this case there are no unit errors in the example, but if `u` and `v` had different units then our solution would not report any error, even though there should be.

There are some difficult situations to cope with. For example, an equation where the units of only some of the variables are known, may be enough to derive the units of the rest. Example below:

$$\begin{array}{lll} a & = & b + c * d \\ a & \text{is} & [m] \\ c & \text{is} & [m/s] \end{array} \quad \Rightarrow \quad \begin{array}{lll} b & \text{is} & [m] \\ d & \text{is} & [s] \end{array}$$

Another complication is that the solution needs to be independent of the order of the equations. A naïve implementation may have to look at equations multiple times. Here is one pathological case I can think of:

$$\begin{array}{lll} a & = & b \\ b & = & c \\ c & = & d \\ d & = & e \\ e & \text{is} & [m] \end{array} \quad \Rightarrow \quad a, b, c, d \text{ is } [m]$$

Chapter 6
Conclusions

We have implemented unit checking in JModelica.org. The unit check, can find inconsistent units in equations, when all variables and constants have a declared unit. With unit inference, which would take some more work, more equations could be checked. Still, with what we have implemented we found 2 errors in the Modelica Standard Library. We ran our solution on all the 366 example models in MSL. Our implementation is used in Modelon's products.

Producing a string from a unit, to be used in error messages, is difficult to do well [2]. There is room for future improvement here.

Bibliography

[1] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L Steele Jr. Object-oriented units of measurement. *ACM SIGPLAN Notices*, 39(10):384–403, 2004.

[2] Peter Aronsson and David Broman. Extendable physical unit checking with understandable error reporting. In *Proceedings of the 7th International Modelica Conference*, pages 890–897, 2009.

[3] Modelica Association. Fmi website. https://www.fmi-standard.org/start accessed 2016-06-08.

[4] Modelica Association. Modelica and the modelica association. https://modelica.org/ accessed 2016-03-18.

[5] David Broman, Peter Aronsson, and Peter Fritzson. Design considerations for dimensional inference and unit consistency checking in modelica. In *Proceedings of the 6th International Modelica Conference*, pages 3–12, 2008.

[6] Bureau International des Poids et Mesures. *The International System of Units (SI)*, 8th edition, 2006. also known as the SI brochure.

[7] Open Source Modelica Consortium. Openmodelica. https://www.openmodelica.org/ accessed 2016-06-08.

[8] JastAdd group. Jastadd. http://jastadd.org/ accessed 2016-06-08.

[9] JModelica.org. Jmodelica.org website. http://jmodelica.org/ accessed 2016-03-18.

[10] Johan Åkesson, Tove Bergdahl, Magnus Gäfvert, and Hubertus Tummescheit. Modeling and optimization with modelica and optimica using the jmodelica.org open source platform. In *Proceedings of the 7th International Modelica Conference 2009*, 2009.

[11] Sven Erik Mattsson, Hilding Elmqvist, and AB Dynasim. Unit checking and quantity conservation. In *6Th International Modelica Conference*, 2008.

[12] Modelica Association. *Modelica - Language Specification*, Version 3.3 Revision 1, July 2014. https://modelica.org/documents/ModelicaSpec33Revision1.pdf accessed 2016-05-16.

[13] Dassault Systemes. Dymola. http://www.3ds.com/products-services/catia/products/dymola accessed 2016-06-08.

Appendices

Appendix A
Unit Expression Grammar

```
unit_expression:
    unit_numerator [ "/" unit_denominator ]
unit_numerator:
    "1" | unit_factors | "(" unit_expression ")"
unit_denominator:
    unit_factor | "(" unit_expression ")"
unit_factors:
    unit_factor [ unit_mulop unit_factors ]
unit_mulop:
    "."
unit_factor:
    unit_operand [ unit_exponent ]
unit_exponent:
    [ "+" | "-" ] integer
unit_operand:
    unit_symbol | unit_prefix unit_symbol
unit_prefix:
    Y | Z | E | P | T | G | M | k | h | da |
    d | c | m | u | p | f | a | z | y
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 3 Juni 2016

EXAMENSARBETE Units of Measurement in a Modelica Compiler
STUDENT Daniel Eliasson
HANDLEDARE Niklas Fors (LTH), Jesper Mattsson (Modelon AB), Jonathan Kämpe (Modelon AB)
EXAMINATOR Görel Hedin

Måttenheter i ett simuleringsspråk

POPULÄRVETENSKAPLIG SAMMANFATTNING **Daniel Eliasson**

Vårt examensarbete handlar om att felkontrollera måttenheter i uttryck och ekvationer i matematiska modeller av fysikaliska system. Till exempel bör man inte addera en längd till en massa eller sätta en hastighet lika med en spänning. Arbetet handlar specifikt om enheter i språket Modelica, och vi har en lösning i JModelica.org.

Modelica

Modelica är ett objekt-orienterat deklarativt språk för modellering av fysikaliska system. Det används till att simulera och optimera bland annat bilar och kraftverk. När man utvecklar fysikaliska system är det bra om man kan testa nya idéer snabbt.

Modeller i Modelica är beskrivna med variabler och ekvationer. Modelica är även ett objekt-orienterat språk. Modeller kan vara uppbyggda av andra modeller vilket tillåter återanvändning av kod.

JModelica.org är en platform med olika verktyg för simulering och optimering i Modelica. Bland annat innehåller den en kompilator som konverterar Modelica kod till en simuleringsbinär, som sedan kan laddas in i andra vertyg som PyFMI (en del av JModelica.org), OpenModelica, och Dymola. Det finns också stöd för att importera till Matlab och Maple.

Kompilatorn i JModelica.org läser in Modelica koden för en modell och bygger upp ett ekvationssystem som sedan optimeras och tillslut bygger den kod för beräkningar som görs under simuleringen. På vägen så kontrollerar den att Modelica koden är korrekt. Det är här vårt projekt kommer in. Genom att kontrollera att måttenheter stämmer överens i ekvationerna.

Enhetsanalys

För att kontrollera att enheter stämmer överens behöver vi först analysera uttrycken i ekvationerna. I bilden är ett exempel på informationsflödet genom syntaxträdet vid enhetskontroll av ekvationen `d = (v1+v2)*t`. Först hämtas enheterna på variablerna. Sen vid multiplikation av två deluttryck multiplicerats enheterna av deluttrycken och vid addition eller likhet mellan två deluttryck måste de ha samma enhet.

Internt i kompilatorn skrivs alla enheter om i termer av SI basenheter. Till exempel $\mathrm{N} = \mathrm{kg\,m\,s^{-2}}$.