Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica

David Broman[1] Peter Aronsson[2] Peter Fritzson[1]

[1]Department of Computer and Information Science,
Linköping University, Sweden, {davbr,petfr}@ida.liu.se
[2]MathCore Engineering, Sweden, peter.aronsson@mathcore.com

Abstract

The Modelica language supports syntax for declaring physical units of variables, but it does not yet exist any defined semantics for how dimensional and unit consistency checking should be carried out. In this paper we explore different approaches and new constructs for improved dimensional inference and unit consistency checking in Modelica; both from an end-user, library, and tool perspective. A proposal for how dimensional inference and unit checking can be carried out is outlined and a prototype implementation is developed and verified using several examples from the Modelica standard library.

Keywords: dimensional analysis, unit checking; dimensions; types; Modelica; language design

1 Introduction

The Modelica language enables expressive modeling by making use of object-oriented acausal constructs. However, certain powerful language constructs easily lead to modeling errors, which are often hard to detect at simulation time. One class of modeling errors that can be detected statically before simulation is model and equation consistency with regards to physical *dimensions*, *quantities* and *units*. The Modelica language specification [12] states how units and quantities can be declared. However, the semantics and strategy for how physical units and dimension of quantities can be checked for consistency, are not described in the specification.

Several of the available tools (e.g., Dymola[4] and Simulation X[8]) implement various algorithms for handling units and dimensions. Furthermore, tool specific language constructs are being added to enable better unit consistency checking. However, this may lead to incompatibility, where some tools reject certain model and others accept them. Unit related research results within the field of programming language (e.g., [1, 5, 9, 13, 18]) have shown that there exist many concepts and constructs that affect the possibility and simplicity to perform correct dimensional and unit checking. Design considerations must be taken from both the *end user perspective* and from the *library and tool implementor perspective*.

This paper introduces and discusses several different concepts and constructs, which are important when designing a language with support for dimensional inference and unit consistency checking[1]. Examples are given using both existing Modelica syntax, and additional suggested constructs. The main contribution of the work is the suggested design for incorporating the unit checking as part of the elaboration (instantiation) process, which supports both implicit inference of unspecified dimensions and rational numbers of dimension exponents. To verify the design, a prototype implementation was constructed in the OpenModelica [17] environment.

The paper is structured as follows: Section 2 introduces fundamental terminology and describes design considerations affecting primarily the end user. Section 3 describes design issues from a library and tool perspective. Both these sections explore the design space in which a specific design can be created. Section 4 specifies a number of design choices made for a prototype implementation created in the OpenModelica environment. Section 5 discusses related work and section 6 concludes the paper.

[1]In the remainder of the paper, the term *unit checking* will be used for *dimensional checking* as well. However, note that even if a system is dimensionally consistent, it might have conflicting units of measure.

2 End User Perspective

In this section, several aspects of unit checking will be discussed primary from an end user perspective. The section starts by refreshing fundamental terminology; followed by description of concepts such as type checking and polymorphism.

2.1 Units, Quantities, and Dimensions

Physical *quantities* are organized into different *dimensions*, such as `length`, `time`, and `mass`. The SI-system [7] defines seven *base quantities*, which can be combined to form new *derived quantities*.
For a particular quantity, there exist several different *units*, e.g., the quantity `length` can be used with both of the units `meter` and `foot`. To convert between different units within the same quantity dimension, *conversion factors* are defined. To convert from `foot` to meter a *scale factor* of 0.3048 is multiplied to the measured value. However, some unit conversions are more complex. For example, the formula $T_{Celsius} = (5/9) * (T_{Fahrenheit} - 32)$ for converting `Fahrenheit` to `Celsius` involves both a scale factor of $5/9$ and an *offset* of value $-32 * (5/9)$.
The SI-system defines seven *base units* (`m,kg,s,A,K,mol,cd`) as well as *derived units*, which are accepted within the SI-system. These derived units have specific names and symbols and always have a corresponding normalized form expressed in base units. For example newton meter has the symbol N m, which has the expression $\text{m}^2\ \text{kg}\ \text{s}^{-2}$.
For some derived quantities, the dimensional exponents are zero. Such a quantity is referred to as *dimensionless* or having dimension one. For example the derived quantity `plane angle` with derived unit `radian` is such a dimensionless quantity.
In Modelica, there is a syntax to define derived unit using base unit expressions. For example, the above expression of newton meter can be expressed as `"m2.kg.s-2"`. From now on, this syntax will be used for describing unit expressions.

2.2 Static Unit Type Checking

When simulating Modelica models, the state of a dynamic model changes during the simulation, but the relation between the units of variables should not change dynamically[2].

[2]Using algorithms and functions, it is possible to define expressions that violates this principle. However, it would require the theory of dependent types to manage this property statically.

Hence, unit and dimensional checking can advantageously be performed statically at compile time. This process is typically accomplished by using a *static type checker*, which takes a Modelica model as input and returns one of three possible answers:

- *Consistent and complete.* The equations, connectors, hierarchy composed components, and the declared derived physical units match without exception. All variables have a specific unit assigned to it.
- *Consistent and incomplete.* The model is consistent (no conflicting constraints), but some variables have no units assigned to them.
- *Inconsistent.* One or several relations mismatch. For example, an equation `a = der(b)*33+c` is inconsistent if `a` and `c` do not have the same units, or if the unit of `b` multiplied by `"s"` (time) is not equal to the unit of `c`.

A language and type checker can be designed to *infer* missing unit types, which can result in both a consistent and an inconsistent result.
Furthermore, from a user's point of view, it is important to *know* that the model is consistent, e.g., that the type checker can *guarantee* that unit errors do not exist. The property that a tool cannot find any inconsistencies in a model, does not imply that the model is consistent. In our proposal, this is a strong requirement for the design of the unit checker.

2.3 Detecting Errors, Isolating Faults

The previous described approach for unit checking enables *detection* of modeling errors, i.e., to give a sound judgement of the model's correctness regarding physical units and quantities. However, even if a tool can respond that a model is incorrect, it is very important for the user to know where in the model the fault is located. Hence, the tools' ability to *isolate faults* in a model is critical for making the unit checking process useable.

2.4 Polymorphism

A language where an object only can be of one type is said to have a *monomorphic* type system. This leads to a very restrictive language, with limited expressiveness. Modelica is a *polymorphic* language, where polymorphic behavior is primarily expressed using subtyping polymorphism.

Consider the following example of the block `Gain`, defined in the Modelica standard block library.

```
block Gain
  parameter Real k(unit="1") = 1;
public
  Interfaces.RealInput u;
  Interfaces.RealOutput y;
equation
  y = k*u;
end Gain;
```

Both input and output to and from the model are defined using `Real` types, i.e., no units are defined for this block. If a unit checker should be able to check instances of this block, unit types must be specified for its formal parameters. For example, both input and output can be defined to have unit type `Voltage`. However, this would result in a new block definition for every imaginable unit, which clearly is impractical.

A solution to this problem which is being implemented in this proposal is the use of *unit type variables*, and so called *parametric polymorphism* i.e., the block is declared to take a unit type variable `'p` as both input and output. Hence, the unit information is propagated from the input to the output[3]. This approach is similar to ordinary type variables used in for example Haskell [16] or Standard ML [11].

For general information about types and polymorphism see [3]. An accessible description on how types are related to Modelica can be found in [2].

3 Design from Library and Tool Perspective

This section presents requirements and a proposed design for unit checking from the perspective of implementers of libraries and tools.

3.1 Unit Type Declaration

There are two approaches of handling declaration of unit types, *implicit* unit type inference or *explicit* type declaration.

- Implicit type inference means that the user does not specify units for all variables and that the tool uses type inference to deduce the units of those variables.
- Explicit type declaration means that the user specifies units for variables, and thus removes the need of deducing units.

For instance, consider the following example:

```
model A
  Real(unit="m") x1,y1,d1,d2;
  Real x2,y2;
equation
  d1 = sqrt(x1^2+y1^2);
  d2 = sqrt(x2^2+y2^2);
end A;
```

The example calculates the distances of two points to the origin `(0,0)`. The first point `(x1,y1)` uses explicit unit type declaration, giving `x1`,`y1` and `d1` the unit `"m"`, and the second point `(x2,y2)` uses implicit type inference, where units are not specified. In the second case the units can be deduced from the unit of the distance variable `d2`, i.e., the unit type of `x2` and `y2` are *inferred* from the unit type of `d2`.

A problem is how to distinguish between dimensionless units and implicit type inference. Consider the following declaration:

```
  Real x;
```

Is `x` dimensionless or should the type be inferred (i.e., has *any* dimension)? The most probable interpretation is that it should be inferred. There are several alternatives of how to declare a dimensionless unit. One solution is to use

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

It is important to differentiate between any dimension and dimensionless, because the distinction can give better information for the unit checker to perform its task.

To be able to handle parametric polymorphism it must be possible to declare *unit type variables*. A unit type variable can hold any unit type and thus provides flexibility of e.g., writing functions. For instance, consider the following example:

```
function myDer
  input Real x(unit="'p");
  output Real y(unit="'p.s-1");
algorithm
  y:= der(x);
end myDer;
```

The example is a wrapper around the `der` operator. The unit of the input argument uses a unit type variable `"'p"` which is used to express the unit of the result from the function. Here the character `'` is part of the type variable identifier and indicates that this is a type

[3]Note that parameter k needs to be explicitly defined to be dimensionless (unit="1") in order to make a unit type inference algorithm to work. If it was left as unspecified, the gain could generate any possible unit, regardless of its input.

variable and not a normal variable. Using a type variable makes it possible to use the `myDer()` function for any type of unit, and still being able to express the relation between the unit types of the input and output argument.

3.2 Unit Conversion

For many situations it is necessary to convert expressions from one unit to another. A unit conversion does not change the dimension of an expression, only its value. For instance:

```
SI.Length d1 = 25.4;
Real d2 =
   unitConvert(d1,"mm");
```

For this case 25.4 is interpreted as meter (defined in `SI.Length`). The proposed built in function `unitConvert(var,unit)` converts the value to 25400 and assigns it to `d2`. Moreover, d2 is now assumed to have unit `"mm"`. Note that it is not possible to just scale this using an ordinary multiplication, since the user must tell the type checker that the unit has been changed.
In conclusion, unit conversion is a fundamental requirement to be able to work conveniently with units.

3.3 Representation of Units

The unit checking mechanism requires the tool to be able to distinguish between different (base) units. This is typically solved (e.g., in [14, 15]) by having a vector of seven base units, as described by the SI standard [7]. For instance, energy can in the SI units be described using `"J"` (Joule) or `"N.m"` (Newton meter) corresponding to the base unit `"m2.kg.s-2"`. Currently, a Modelica tool would need to know that `"J"` or `"N.m"` correspond to the base unit `"m2.kg.s-2"` and how to construct the appropriate vector for such a unit.
To be able to handle functions like calculating the square root of a value (the sqrt function), the coefficients of the dimension vector must be able to handle more than integer numbers. By using rational numbers instead it is possible to express e.g., the square root with exponent (1/2). Note that it is not possible to use floating point precision as coefficients , since that would lead to roundoff errors.
A problem related to the representation of units is how to present a unit to the user. Often a user has no idea what the unit `"m2.kg.s-2"` means. Instead, the user expects the derived unit to be output, i.e., `"N.m"`. The problem of unparsing (pretty printing) the internal unit representation to a string must be considered. Often, the choice of derived units to use is not obvious, and heuristics must be used to achieve what a user might expect as output. Such heuristic is not trivial to do and it might even be different depending on the context (application area) of the user model.

3.4 Defining Units in the Modelica Language

To be able to handle other units than those described by the SI standard, a more elaborate design than using seven base units must be introduced. For instance, a financial institute involved in modeling and simulating the stock market might be interested in using the quantity `"money"`. Also, they would like to be able to add scaling factors between different units of money ($, €, SEK, etc.). Thus, an important design requirement for the unit checking framework is that the number of base units is not known a priori, i.e., end users must be able to add whatever units they want. Also, the scale and offset information must be available for the unit checking module. Finally, it must also be possible to describe the relation between base units and derived units.
Currently, Modelica does not have support for adding scaling (and offset) for units, neither can one add ones own "base units". Today, Modelica has some knowledge about the SI units, e.g., a Modelica tool with unit checking capabilities knows that `unit="m"` refers to the base unit meter and `unit ="F"` refers to the non-base unit farad (expressed as `"m-2.kg-1.s4.A2"` in base units) and not to Fahrenheit. But, if users should be able to add their own base units, the language should instead be extended so that base units can be described in Modelica. The SI-units package would then first declare the SI base units, and then derive units based on these base-units.
Moreover, information for converting between units is not covered by current Modelica. To be able to convert between different units, scaling and offset information must be introduced. For instance, consider converting between Fahrenheit and Kelvin. This can be achieved using a scaling factor and an offset as illustrated by the conversion function in the standard library:

```
function from_degF
  input  NonSIunits.Temperature_degF
         fahrenheit;
  output Temperature kelvin;
algorithm
  kelvin := ((fahrenheit - 32)*5)/9 -
            Modelica.Constants.T_zero;
end from_degF;
```

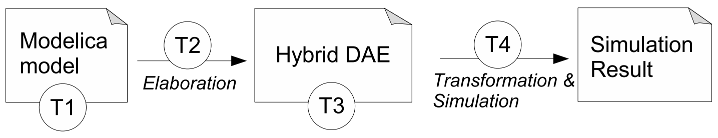

Figure 1: Possible unit checking-times (T1,T2,T3,T4) during the Modelica compilation and simulation process.

If the scale and offset information instead is added to the unit types (e.g., as attributes to the built-in Real class), such conversion functions would not be required. Instead the tool could perform the conversion using the built-in `unitConvert()` function, rendering convert functions in the standard library redundant.

3.5 Time of Checking

There are several different points in time during the translation process where the unit checking mechanism could be introduced, see Figure 1.

- T1 - At the model level.
- T2 - During elaboration.
- T3 - At the hybrid DAE (flat Modelica) level.
- T4 - During runtime/simulation.

Some checks can be made at the model level (T1), performing checks for each individual sub-model. Local equations in the model can be checked this way, but not equations generated from connecting components together, or components where types must be deduced from the surrounding environment (e.g., connections or modifiers). Another approach is to combine the unit checking phase with the elaboration (flattening) process (T2).
Checking on the flat model (T3) is of course feasible, leading to a large check of the overall system. The advantage of this approach is its simplicity; a translation of the model into equations for the unit checking module is performed only once. The disadvantage is that it is much harder to isolate the fault, since only the flat set of equations is available. Also, this approach will not make use of already checked parts, e.g., checking the model equations of an electrical resistor will be done not only once but for as many times as the resistor model is used as a component. The gPROMS unit checking tool [14, 15] uses this approach. Finally, some analysis cannot be performed statically and must then be performed during runtime, i.e., during the simulation (T4).

4 Prototype Implementation

A prototype implementation based on the design requirements presented above is under development in the OpenModelica[17] and MathModelica[10] compilers. The compiler does a static (during compilation) check of dimensions and units of measure.

4.1 Design

The design includes the following aspects:

- Rational numbers as exponents on dimensions.
- Unit type variables in declarations.
- Literal constants are treated differently depending on context (dimensionless in multiplication/-division and unknown in addition/subtraction).
- Type inference of dimensions.
- User defined base and derived units.
- Checking is performed during elaboration / flattening to enable better fault isolation.

The design is split into separate parts, see Figure 2.
One part is integrated with the elaboration (flattening) process in the OpenModelica compiler. It will create an equation systems to be solved by the Unit Checker (the second part) for model components according to the same principles as components are instantiated in Modelica (i.e., a recursive process). This is done by first adding units to a unit store by calling the addStore function in the UnitASTBuilder module. Next, local equations are traversed to build unit terms, with the buildTerms function. Both the unit store and unit terms are defined in the UnitAbsyn module. Finally, the check function in the UnitChecker module is called to perform the dimension analysis. The result from the checking of each component contains two pieces of information. First, for each component it will receive an answer whether a component is Ok (consistent and complete), inconsistent (incompatible types) or consistent and incomplete (not enough information available). Secondly, it will calculate the resulting unit type variables of a component which can then be used

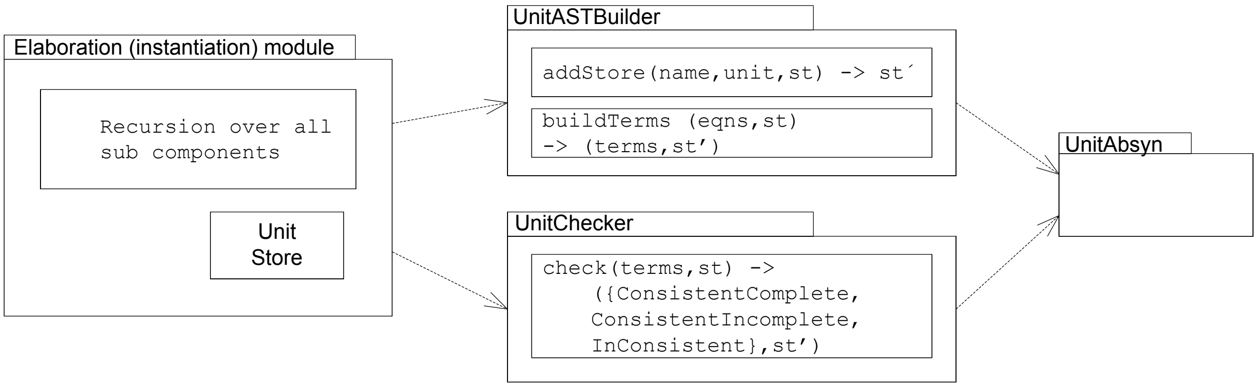
Elaboration (instantiation) module
Recursion over all sub components
Unit Store
UnitASTBuilder
addStore(name,unit,st) -> st´
buildTerms (eqns,st) -> (terms,st')
UnitChecker
check(terms,st) -> ({ConsistentComplete, ConsistentIncomplete, InConsistent},st')
UnitAbsyn

Figure 2: Outline of the main modules of the unit type checking engine of the prototype implementation. Arrows describe dependencies between modules.

when checking the complete model. This will give the following steps of the unit checking function.

1. Check components in the class.
2. Build a new equation system from the type variables from each component together with local equations and connections.
3. Call the unit checker for the model itself.

Note that checking the components of a class means a recursion over the three steps for the class of the component.

The equation systems for the unit checker are created from two data structures, a unit store that holds units of variables, and unit terms that describe constraints between different variables. The following sections show how these are built.

4.1.1 Storing Units

Each variable in a model has a corresponding unit. A unit can be

- A specified unit, e.g., `"m/s"`.
- A unit type parameter e.g., `"'p"`, with an optional exponent, e.g., `"'p^2"`.
- A combination of specified unit and type parameter, e.g., `"'p/s"`.
- unspecified unit e.g., the unit of a declaration `"Real x;"`.

The unit store is a data structure that holds the units of variables. It gives a mapping from a variable name to its corresponding unit. During the instantiation and unit checking process the unit store is updated with new units. The following model shows how the unit store is used:

```
model SimpleOde
  Real x;
  Velocity v;
equation
  der(x)=2*v + 1.0;
end SimpleOde;
```

First the unit store is built by adding the units of the variables `x` and `v`. Since `x` is declared as a `Real` it gets an unspecified unit, and `v` gets the unit `"m/s"`. After the unit checking module has been executed on this class, it will update the unit store with the unit for `x` with `"m"`, because this was inferred by the UnitCheck module. This information can then be used higher up in the instance tree to check units of other components.

4.1.2 Building Unit Terms

The second data structure required for building unit constraint equations is the Unit Term which describes relations between variables. This structure is similar to the data structure for equations, containing nodes for e.g., addition, multiplication, etc. It is sufficient to only have four types of relations between units: multiplication of terms, division of terms, addition of terms, and equality between terms. Since an addition of two variables and a subtraction of two variables both imply the same rules for the units, both of these can be expressed using the same unit term. The leaf nodes of terms are references to units in the unit store.

Let us again consider the example SimpleOde above. We use ADD and MUL for addition and multiplica-

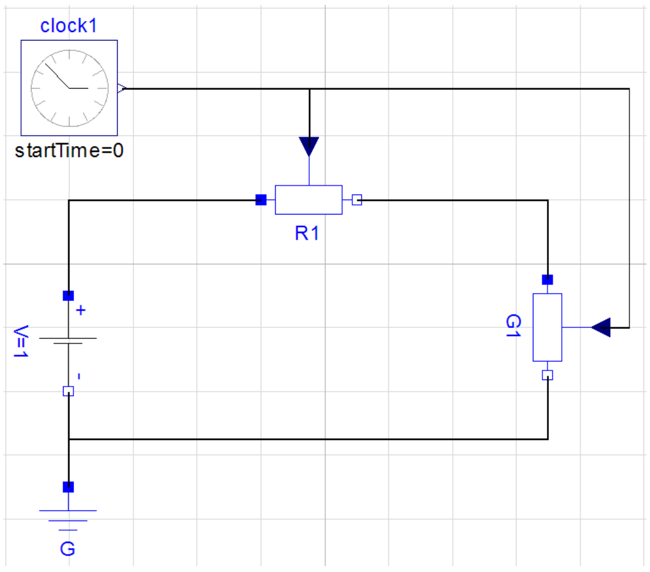

Figure 3: An inconsistent circuit that should fail during dimensional checking.

tion in our data structure and EQN for equality between terms. For the leaf nodes, with references to the unit store, are described with LOC. The example above corresponds to the following terms (somewhat simplified):

```
EQU(
  LOC("der(x)"),
  ADD(
    MUL(LOC("V"),LOC("2")),
    LOC("1.0")))
```

From the unit store and the unit terms, constraint equations are built. A multiplication of unit terms means that the unit vector is added, and an addition of unit terms means that the units must be equal.

4.1.3 Built-in Functions and Operators

The built-in functions and operators are extended with units containing unit type parameters. That gives us a uniform way of dealing with functions, regardless if the function is a built-in function, a built-in operator, or a user-defined function. For instance, the der operator is internally described as

```
function der
  input Real x(unit = "'p");
  output Real y(unit = "'p/s");
  external "builtin";
end der;
```

That is, applying the derivative operator to an expression will change its unit by multiplication with "s-1".

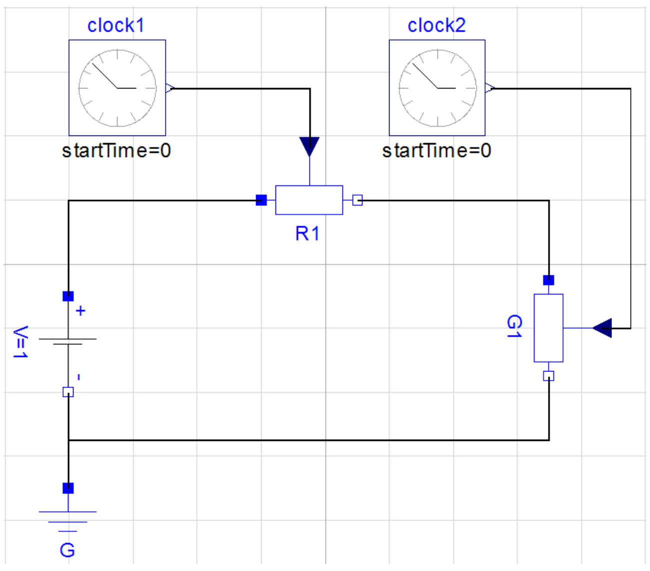

Figure 4: A dimensionally correct circuit.

4.2 Example

Let us consider an example using components from the Modelica Standard Library to illustrate the different aspects of unit checking. Figure 3 shows an example where unit checking will return an error because of inconsistent units[4]. A VariableResistor and a VariableConductor is fed from the same signal source, taken from the Blocks library. All sources in the Blocks library have unspecified units, such that they can be used in any context. The unit checker will find that the unit of the output of the clock generator should be both "Ohm" (Resistance) and "S" (Conductance), i.e., an inconsistency is reported. This inconsistency is detected first when the local equations of the Circuit model is unit type checked. The unit store then contains an unspecified unit for the clock generator (clock1.y) and specified units for the inputs on the resistor R1 (R1.R) and the conductor G1 (G1.G).

To resolve the inconsistency of the circuit the user has to use two separate clock generators, see Figure 4. The unit of clock1.y will become `"Ohm"` and clock2.y will become `"S"`, resulting in a consistent system.

When using math blocks (Gain, Add, TransferFunction, etc) in models it becomes evident that polymorphism is required. For instance, lets add a gain to

[4]The circuit is inconsistent since the VariableResistor and VariableConductor have declared their inputs to Resistance and Conductance respectively. If they were declared as dimensionless the circuit would have been consistent, thus also making it a library design issue. Also, it could be possible to have different unit checking semantics depending on the causality of equations, which would allow this kind of connections.

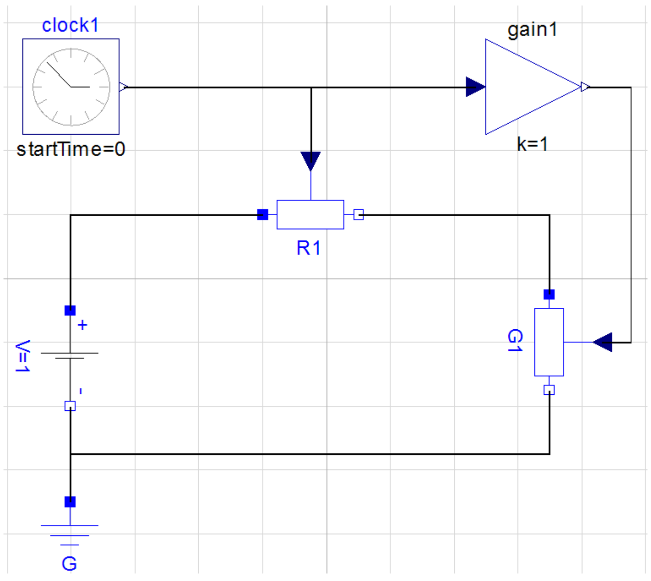

Figure 5: An inconsistent model with a polymorphic block.

our inconsistent model, see Figure 5. The gain block should be possible to use for any unit, i.e., it should be a polymorphic block. If that would not be possible, the user would have to write a new block model for each particular use, in this case for amplifying a Conductance signal. In our implementation, the unit checker will treat the Gain block as having a polymorphic unit and assign a unit type parameter to it. The result of checking the gain block is a unit type parameter that propagates the unit of the input to the unit of the output. Hence, when the circuit model is checked, the unit from the VariableConductor is propagated to the unit of `clock1.y`, leading to an inconsistent system of equations. Typically, for larger block models, this propagation can be performed over many subsystems of components. This implementation will however lead to a detection of the inconsistency at the lowest level possible, making it easier for the user to correct the inconsistency.

5 Related Work

Unit checking has been introduced in several Modelica tools over the last couple of years, for instance, Dymola[4] from Dynasim and Simulation X[8] from ITI GmbH. Dymola version 6.1 has a unit checking mechanism, as well as support for deduction of units. However, unit parametric polymorphism is yet not supported.

Simulation X has a conversion extension to Modelica for giving units to literals. For instance, the expression `a + 2.5 'mm'` will translate the literal `2.5` into SI base unit meter by multiplying it with `10e-3`.

Both of these tools will (or soon will) support entering other units than default units for e.g., parameter values, i.e., making it possible to enter `2.5 mm` as a parameter value. The displayUnit attribute of Modelica standard is available for this purpose.

Unit checking and checking of dimensional inconsistency has been extensively explored in the programming language research community and is far from a new research area. Many library-based approaches exist for imperative programming languages, such as a package approach for Ada [6] and a template approach in C++ [18]. An approach for dimensional inference is presented in [19], where gaussian elimination is used for solving the resulting equation system. The work shows how dimensions with rational exponents can be added to the simply typed lambda calculus.

In Kennedy's thesis [9], an extension of a core calculus of ML with support for type inference over dimension types is given. Lately, dimension and unit checking have also been addressed in a nominally typed object-oriented language [1].

Besides the work on gPROMS [14, 15], few attempts have been made to incorporate dimensional and / or unit checking in equation-based object-oriented languages, such as Modelica. In addition, even though Modelica today supports syntax for stating units of variables, no sound solution exists that guarantees the absence of unit errors.

6 Conclusions

This paper has presented a design for dimensional analysis and unit checking of Modelica models. Requirements from an end user and tool perspective have lead to a design which has been implemented as a prototype on top of the OpenModelica and MathModelica compilers. MathModelica has also been used for building the models presented in this paper, and a future release of MathModelica will contain unit checking based on the design in this paper. The design introduces unit type variables enabling polymorphism of unit types in Modelica, which increase the safety and flexibility of the dimensional analysis. We have also chosen to represent exponents as rational numbers which enables dimensional checking of e.g., the sqrt function. The design of the dimensional analysis also allows the possibility of adding additional base units, on top of the seven base units of the SI system. This enables modeling of e.g., financial systems using base unit money, and other application areas.

The prototype implementation has been described and illustrated with several examples from the standard library. The analysis results in either a consistent and complete system, a consistent but incomplete system (which means that not sufficient unit information is available to fully determine units) or an inconsistent system (indicating where the inconsistency is located). By using the prototype we have detected some minor problems with the standard library. For instance, the Gain component in the Blocks Math library currently has unspecified units on its gain parameter. In order to fully check the dimensions of models using this component, the gain parameter should be dimensionless.

This paper has also discussed unit conversion, even though this has not yet been implemented. Nonetheless, some ideas presented here could be a useful starting point for the Modelica Design Group's activities regarding this topic.

Acknowledgments

This research was funded by CUGS (National Graduate School in Computer Science), MathCore Engineering, by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project, the Swedish Research Council (VR), and Forska&Väx program under the project Lättanvänt Generellt Simuleringsverktyg för Industriell Produktutveckling.

References

[1] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Jr. Guy L. Steele. Object-Oriented Units of Measurement. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 384–403, Vancouver, BC, Canada, 2004. ACM Press.

[2] David Broman, Peter Fritzson, and Sébastien Furic. Types in the Modelica Language. In *Proceedings of the Fifth International Modelica Conference*, pages 303–315, Vienna, Austria, 2006.

[3] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[4] Dynasim. Dymola - Dynamic Modeling Laboratory (Dynasim AB). `http://www.dynasim.se/` [Last accessed: Jan 22, 2008].

[5] Narain Gehani. Ada's derived types and units of measure. *Software Practice and Experience*, 15(6):555–569, 1985.

[6] Paul N. Hilfinger. An ADA Package for Dimensional Analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, 1988.

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

[8] ITI. SimulationX. `http://www.iti.de/` [Last accessed: November 8, 2007].

[9] Andrew Kennedy. *Programming Languages and Dimensions*. PhD thesis, St. Catharine's College, University of Cambridge, UK, UK, 1996.

[10] MathCore. MathModelica System Designer: Model based design of multi-engineering systems. `http://www.mathcore.com/products/mathmodelica/` [Last accessed: Jan 23, 2008].

[11] Robin Milner, Mads Tofte, Robert Harper, and David MacQuee. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[12] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.0*, 2007. Available from: `http://www.modelica.org`.

[13] Gordon S. Novak. Conversion of Units of Measurement. *IEEE Transactions on Software Engineering*, 21(8):651–661, 1995.

[14] Daniel Persson. Dimensional Analysis and Inference for gPROMS. Master's thesis, Department of Computer Science and Engineering, Mälardalen University, Sweden, 2003.

[15] Mikael Sandberg, Daniel Persson, and Björn Lisper. Automatic Dimensional Consistency Checking for Simulation Specifications. In *SIMS 2003*, September 2003.

[16] Simon Peyton Jones. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[17] The OpenModelica Project. `http://www.ida.liu.se/~pelab/modelica/OpenModelica.html` [Last accessed: January 22, 2008].

[18] Zerksis D. Umrigar. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices*, 29(9):135–139, 1994.

[19] Mitchell Wand and Patrick O'Keefe. Automatic Dimensional Inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic - Essays in Honor of Alan Robinson*. The MIT Press, 1991.