Implicit Unit Conversion in Modelica

Henrik Tidefelt[1] Quentin Lambert[1]

[1]Wolfram MathCore, Sweden, {henrikt,qlambert}@wolfram.com

Abstract

There are many situations in which a Modelica model needs to handle quantities which are not expressed in the often preferred unscaled SI units. Applying correct unit conversions is extremely important in such situations, and the risk of human error needs to be mitigated using unit-aware technology. Considering the power of unit checking mechanisms in several Modelica tools today, one can be surprised that unit conversion in Modelica still needs to be performed using error-prone user-written formulas and functions. It is demonstrated how automatic and implicit unit conversion can be introduced in Modelica, and that this can be done safely. The benefits of this approach are illustrated in a variety of examples and applications.

Keywords: unit checking, unit conversion, Modelica

1 Introduction

1.1 Why model with units?

Units of measurement, in this work simply referred to as *units*, play an important role in the modeling domains where Modelica is used for several reasons. An often cited reason is that attention to units of measurement would have avoided some spectacular engineering failures in the past due to mixing up numbers in metric units with numbers in non-metric units. Another reason is that use of units of measurement allows *dimensional analysis* of programs, allowing certain errors in expressions and equations to be detected during model translation. Units of measurement also enrich the presentation of computed results, and give hints regarding the kind of quantity that computed numbers represent. A key convenience of working with unit aware tooling is also the possibility of separating the units used in numeric computation and storage of results, from the units used to conveniently enter parameter values and display results.

1.2 The use of unit conversion in Modelica

If all units in the world of Modelica models were equivalent to unscaled SI base units, there would be no reason to have support for unit conversion. However, many Modelica models exist in a context where also other units are used, and in such tool chains there needs to be proper support for keeping track of and converting between units in order to avoid the sometimes subtle but catastrophic consequences of interpreting numeric values with an incorrect understanding of the associated units. In a complex environment where a Modelica model is interfaced with surrounding systems, it is hard to imagine a better place for performing the necessary unit conversions than inside of the Modelica model – it would be unreasonable to expect that all of the surrounding systems should be capable of adapting their interfaces to a choice of units governed by the Modelica model.

1.3 Unit conversion in Modelica today

There are two common approaches to performing unit conversions in Modelica models today, namely unit conversion factors (possibly embedded in gain blocks) and unit conversion functions. The Modelica Standard Library contains a selection of unit conversion functions as well as examples of using gain blocks with conversion factors.

The approach of using a dedicated function for every supported unit conversion does not scale well to the general task of performing conversion between any pair of conversion compatible units.

With the conversion factor approach, conversions are often created as needed instead of being provided as a collection of predefined conversions. This makes the method of using `displayUnit = "1"` a viable technique for manually setting the correct value of the conversion ratio – the correct value is 1 when expressed in the unit `"1"`, reflecting that there is no scaling in terms of value of quantity:

```
  Real x(unit = "m");
  Real y(unit = "yd");
  final constant Real m_per_yd(
    unit = "m/yd", displayUnit = "1")
    = 0.9144;
equation
  x = m_per_yd * y;
```

However, when a conversion factor is used in equations like above there is a high risk that the value will rarely be seen expressed in the display unit, making it a potential source of error.

On the other hand, when the conversion factor is embedded in a gain block, the gain block with a gain displayed as 1 may appear as unnecessary clutter to an untrained eye, or to someone who is not familiar with the need for explicit unit conversion in Modelica. If the `displayUnit` is removed, the gain block might appear less like unnecessary clutter due to no longer displaying a gain of 1, but at the cost of no longer making it obvious that the block is merely performing a unit conversion, not actually scaling the value of quantity.

1.4 State of the art

General purpose computer algebra systems with unit handling tend to allow implicit unit conversion. For instance, all of the following computer algebra systems support adding values of quantity expressed in different units: Wolfram Language, Maple (through the Units package), Python (through the Pint package), Julia (through the Unitful.jl package), and JavaScript (through the Math.js library). Further, none of the systems allow adding, say, a length to a numeric value. (*Wolfram Language – Quantity* 2025; *Maple – Tutorials Part 10: Units* 2025; *Pint – Tutorial* 2025; *Unitful.jl – Conversion/promotion* 2025; *Math.js – Units* 2025)

An interesting feature of Unitful.jl which is particularly relevant for the current work is the `FixedUnits` concept. It is a mechanism for locally taking control over the implicit unit conversion by disabling it.

Likewise, in systems for causal modeling, the widespread Simulink software offers implicit unit conversion between connected blocks. There is also a block for explicit unit conversion, but no block for getting automatic unit conversion at a particular place in a model.(*Simulink – Converting Units* 2025)

On the more formal side, Kennedy (1997) argues that one reason for annotating programs with units of measurement rather than physical dimensions is that it enables compilers to automatically inject unit conversions.

When it comes to Modelica, unit handling is less developed, explained partly by the added complexity that comes with not having prescribed causality of computations, but probably partly also due to the difficulties caused by not having a syntax for attaching units to numeric literals. Early developments include Mattsson and Elmqvist (2008) and Aronsson and Broman (2009), but to date the only thing which has been formalized in the Modelica language is the string syntax for expressing units. That said, the use of *display units* is well established, allowing values of a variable to be displayed or entered in a different unit than the unit used for the underlying numeric representation. This means that existing Modelica tools already have capabilities for unit conversion, but these capabilities are not leveraged by the Modelica language.

Several Modelica tools implement unit checking with unit inference, and even though implementation details differ, there are no major controversies regarding whether a model is using units consistently or not. It is an ongoing work to standardize unit checking in Modelica, so that a model considered consistent by one tool will also be considered consistent by other tools. To our knowledge, no Modelica tool is currently supporting a non-standard extension for allowing automatic or implicit unit conversions.

1.5 Limitations

This work does not elaborate the handling of units with offsets – typically temperature units – for which there is a difference in the interpretation of a unit depending on whether a value of quantity is relative or absolute. The limitation is a matter of presentation; the unit handling framework in which the current work has been implemented also supports the distinction between relative and absolute values of quantity, but a discussion of its design in this regard would overshadow the cetral topic of this work.

The unit symbols required to be recognized according to the Modelica specification are not even sufficient to support the Modelica Standard Library – for instance, the unit of the `Pressure_bar` type is the "non-standard" (that is, not supported by the Modelica specification) `"bar"`. In order to more closely relate to real world applications with need for unit conversion, examples in this work also make use of non-standard units such as `"lb"`. All of these and many more are supported out of the box by the computer algebra systems mentioned in subsection 1.4, and the authors believe that Modelica one way or another will need to develop in the same direction. In the meantime, there are several ways in which a Modelica tool could provide support for non-standard units. For example, the unit handling framwork used here supports all units appearing in the Modelica Standard Library and a conservative selection of additional well established units out of the box, and also supports per-library customization through vendor-specific Modelica annotations. It is a limitation of this work that the question of how to ensure a consistent selection of non-standard units across tools is not addressed, but the proposed ideas for unit conversion would remain relevant also with only current Modelica's standard units since the standard supports the use of unit prefixes (as in `"mm"`) as well as several unit symbols for time and volume. For example, `"L/min"` is a standard Modelica unit which might require conversion to, say, `"m3/s"`.

2 Notation and preliminaries

2.1 Notation

Concrete units will be written in the style of their Modelica string representation. For example, `"m"` represents the meter, and `"1"` represents the unit of a dimensionless number.

A *numeric value* refers to a real number without unit. For instance, 1.5 is a numeric value. Numeric value is not the same as dimensionless number, as the latter has a unit of `"1"`.

The combination of a numeric value and a unit is denoted a *value of quantity*. (In other contexts, this may also be referred to as a *value* or a *quantity*, but both these terms already have conflicting established meanings in the Modelica context.)

Objects with unit (values of quantity, variables, expressions, etc) belong to a *quantity space*, which is either an *affine space* or *vector space*. Values of quantity in affine space are often said to be *absolute*, while values in vector space are sometimes said to be *relative*.

DOI
10.3384/ecp21827

A numeric literal is said to have empty unit.

2.2 Simple non-standard operators for unit handling

In order to not distract from the central topic of this work in the following sections, a few simple non-standard operators for unit handling will be taken for granted.

2.2.1 Construction of value of quantity

The non-standard expression `withUnit(`x`, `u`)` constructs a value of quantity with numeric value x and unit u. The expression x shall have empty unit.

Example:

```
Real x = withUnit(1.0, "m");
```

When `withUnit` is used with literal operands, the expression is referred to as a *unitful literal*.

There are several appealing alternatives for using special syntax instead of calling the operator by its name as above. In this work, a unitful literal may be constructed by providing the unit in the form of a quoted identifier following the numeric literal:

```
Real x = 1.0'm';
```

2.2.2 Fixed unit conversion

The expression `inUnit(`v`, `u`)` is used to express the value of quantity v in the unit u. The expression v shall not have empty unit. The unit u shall be given by a constant expression and be conversion-compatible to the unit of v. The `inUnit` is an affine function of its first operand, and a tool can inline calls early to reduce the need for symbolic processing rules.

Example:

```
Real x(unit = "m") = 1.0;
Real y(unit = "yd") = inUnit(x, "yd");
```

This operation only uses the same sort of information that Modelica tools need to possess in order to support display unit conversions, with the difference that the information is needed during model translation rather than when editing values or plotting results.

2.2.3 Extraction of numeric value

The expression `withoutUnit(`v`, `u`)` obtains the numeric value of the value of quantity v expressed in the unit u. The expression v shall not have empty unit. The unit u shall be given by a constant expression and be conversion-compatible to the unit of v. The returned numeric value has empty unit.

For example, this can be used to make a unit-safe function for an empirical relation which only holds for a particular choice of units, while the function interface is free to use a more common choice of units:

```
function empirical
  input Real u(unit = "kg");
  output Real y(unit = "kg") =
    inUnit(y_lb, "kg");
protected
  Real u_lb = withoutUnit(u, "lb");
  Real y_lb =
    someEmpiricalRelationInPounds(u_lb);
end empirical;
```

3 Unit checking machinery

This section gives a very brief overview of the constraint-based approach to unit checking which is the foundation for the extended semantics proposed in this work. It is based on ideas found in Hindley-Milner type systems.(Hindley 1969; Milner 1978) The description in this section describes the machinery without support for implicit unit conversion, currently in use in Wolfram System Modeler for checking unit consistency in agreement with common (still not standardized) understanding of how units should be used in current Modelica. A more detailed description of the machinery is outside the scope of this work, but is available in the form of a pull request within the Modelica Change Proposal MCP-0027 *Unit checking* (currently in the state *In Development*).(Lambert and Tidefelt 2024) Although currently not standardized, the unit semantics resulting from the machinery presented in this section will be denoted the *current semantics*. In subsection 6.2, it will then be shown how the machinery can be extended to support automatic and implicit unit conversion.

In addition to the unit of an expression, the machinery also keeps track of the quantity space. Similar to unit checking, there are rules for quantity space checking, and the rules may be used for inference of quantity space. While details of the quantity space handling are outside the scope of this work, it should be mentioned that it is largely independent of the handling of units. In particular, the constraint-based approach to unit checking described in this work is carried out without modification also when quantity space is considered, but it must be remembered that unit inference does not consider – and hence does not determine – unit offsets. For vector quantity space, unit offsets are irrelevant, and hence units inferred by unit checking are fully accurate. For affine vector space on the other hand, unit offsets matter, and in principle it is a responsibility of quantity space checking to determine them. Alternatively – and this is by far the most common situation and corresponding to the limitations of this work – if an inferred unit has zero offset and all conversion compatible units also have zero offset, then knowing the quantity space is neither needed to conclude that the correct offset would be zero in case of affine quantity space, nor to perform correct unit conversion. Hence, it would be needlessly strict to make unknown quantity space an error in this situation, and it can be seen that quantity space checking can be safely ignored as long as units with offset are not considered.

Unit checking is carried out by solving a set of unit *equivalence compatibility* constraints. The unknowns are a combination of unit variables associated with variables in the model, and auxiliary variables introduced during the

process of collecting the constraints. A constraint is a relation between two unit *meta-expressions*, a very simple expression language with operations for multiplication, raising to a power, and differentiation.

In order to make unit inconsistency an error according to the Modelica specification, it needs to be a tool independent property of a model. To this end, the collection of unit constraints takes place at an early stage of model translation where expressions and equations still have a close and dirrect correspondence to the original Modelica input. (Alternative approaches to unit checking applied at later stages of model translation might be easier to implement, but be difficult to define in such a way that unit consistency would become a tool independent property of a model.)

The literals of the meta-expression language, denoted *meta-literals*, consist of the *concrete* units, the empty unit, and the undefined unit. The concrete units correspond to well-formed Modelica unit strings. A constraint is trivially satisfied if either side is empty or undefined, and the inference will never determine a unit variable as empty or undefined. Hence, after completed unit inference, each unit variable will either be determined as a concrete unit or left undetermined.

The notation $\texttt{var}_\texttt{u}$ is used for the unit variable associated with the model variable `var`.

4 Shift of mindset

4.1 The unit equivalence mindset

In current Modelica there is an unstated mindset based on unit equivalence. For example, it is generally expected that this model shall pass unit checking:

```
Real x(unit = "J/s");
Real y(unit = "W") = x;
```

That is, the unit of `y` (`"W"`) does not need to be written identically to the unit of its binding equation (`"J/s"`).

Adding expressions with equivalent units comes down to just adding numeric values:

```
Real x(unit = "J/s") = 1.0;
Real y(unit = "W") = 2.0;
Real z = x + y;
```

It suffices to assume that there is no unit inconsistency in order to conclude that the numeric value of `z` is 3.0. Units only serve as complementary information to the numeric values, but have no impact on the computation of numeric values. A unit for `z` can be determined by unit inference, and it is not perceived as a problem that the exact form of the unit is not uniquely defined; knowing that the inferred unit for `z` will be equivalent to both `"J/s"` and `"W"` is enough.

When all expressions have units, and units are used consistently, mathematical field axioms are fulfilled. For example, let

```
Real x(unit = "s");
Real y(unit = "s");
Real z(unit = "s");
Real u(unit = "m");
```

Then the following properties hold:

- Commutativity: `x + y = y + x`
- Associativity: `(x + y) + z = x + (y + z)`
- Distributivity: `u * (x + y) = u * x + u * y`

When empty literals are present, there is a tradeoff between convenience and possibility of detecting unit inconsistencies. Standardization of unit checking is pointing in the direction of avoiding the "wildcard effect" of literals in multiplicative expressions:

```
// OK, 1.0 effectively has unit "s":
Real a(unit = "s") = x + 1.0;
// Error, 1.0 effectively has unit "1":
Real b(unit = "m.s") = u * 1.0;
```

Unfortunately, this means that multiplication does not distribute over addition in Modelica:

```
Real c = u * (x + 1.0);    // OK
Real d = u * x + u * 1.0; // Error
```

The semantics proposed in this work will preserve the important properties of commutativity and associativity. Likewise, distributivity will be preserved when all expressions have units.

To fix the problem with general distributivity in Modelica will require another approach with is outside the scope of this work, namely broad adoption of unitful literals, so that all remaining literals written without unit can be assigned the unit `"1"`. For example, `u * (x + 1.0's') = u * x + u * 1.0's'`.

4.2 The value of quantity mindset

As long as units are used consistently, the particular choices of units become less important, and it becomes possible to view the relations in the model in terms of related values of quantity rather than the numeric values that would vary with the choice of units. This mindset requires trust in two things:

- Correctness of annotated as well as inferred units.
- That consumers of translated models pay proper attention to units, annotated as well as inferred.

It is of particular importance that a numeric value together with a unit of measurement is understood as just one of infinitely many equivalent representations of the same underlying value of quantity.

To establish the required trust is beyond the scope of this work. However, recent developments in unit checking in common Modelica tools as well as awareness of gained quality of existing Modelica libraries thanks to unit checking support in tools, is expected to build up such trust over time.

DOI 10.3384/ecp21827

5 From error to automatic resolution

This section presents a sequence of models, giving a natural progression from how an equation which is erroneous according to current semantics, eventually can be interpreted as valid thanks to an implicit unit conversion. Details of the implicit unit conversion will be given in section 6.

5.1 Unit inconsistency is an error

A prerequisite for this work is that inconsistent use of units is an error.

```
  Real x(unit = "m") = 1.0;
  Real y(unit = "yd");
equation
  // Prerequisite that this is an error:
  x = y;
```

Thanks to being an error to start with, it will be possible to define semantics without breaking backwards compatibility.

5.2 Explicit unit conversion

The inconsistency can be resolved using `inUnit` for explicit unit conversion:

```
  Real x(unit = "m") = 1.0;
  Real y(unit = "yd");
equation
  x = inUnit(y, "m");
```

5.3 Explicit automatic unit conversion

A non-standard operator named `autoUnit` will be used to denote explicit automatic unit conversion. The operator is similar to `inUnit`, except that the target unit of the conversion is determined from the context.

Let e be an expression of the form `autoUnit(`e_1`)`, and let unit(e) denote the unit of the expression e. Similar to additive operators, a unit variable is conceptually introduced for unit(e). No unit constraints are associated with the `autoUnit` expressions, but all of the following shall hold after completed unit inference:

- unit(e) is a concrete unit.
- unit(e_1) is a concrete unit.
- unit(e_1) is convertible to unit(e).

After completed unit inference, `autoUnit(`e_1`)` is replaced with `inUnit(`e_1`,` unit(e)`)`.

Applying `autoUnit` to resolve the unit inconsistency above:

```
  x = autoUnit(y);
```

5.4 Implicit unit conversion

Finally, as an alternative to explicitly introducing `autoUnit` calls to resolve unit conflicts, it only remains to let the tool implicitly introduce `inUnit` calls where a unit inconsistency between conversion-compatible units would otherwise result. Considering that different ways of resolving a unit conflict using unit conversion may result in different interpretation of empty unit expressions as values of quantity, it must be ensured that implicit unit conversion does not cause any actual or apparent ambiguity of computed values of quantity. How to ensure this is the topic of section 6.

Now, the equation originally considered to be in error is instead interpreted as having implicit unit conversion:

```
  x = y;
```

6 Well-defined values of quantity

This section describes how the unit checking machinery can be extended to support implicit unit conversion without introducing ill or even poorly defined values of quantity. In order to serve as a viable extension of current Modelica, the design will meet the following criteria:

- No change of semantics for models free of unit errors according to current semantics.
- No actual or apparent ambiguity of computed values of quantity.
- Preserved associativity and commutativity of addition.

The first of these items is obtained by only allowing implicit unit conversion where the current semantics would have detected a unit inconsistency. Therefore, existing Modelica programs free of unit inconsistency will remain unaffected by the introduction of implicit unit conversion in the language.

The last two items are related, and will be addressed in the following.

6.1 The potential source of ambiguity

When a potential unit inconsistency is resolved using `inUnit`, it is important to be aware that different choices may lead to different interpretations of empty unit expressions. Consider the following expression, without allowing implicit unit conversion:

```
1.0'cm' + 0.5 + 1.0'm'
```

The expression has a unit inconsistency which can be resolved using unit conversion. Here, the conflict will be resolved using `inUnit`. It would also be possible to use the more convenient `autoUnit`, but this might have obscured the fact that the problem is unrelated to the automatic choice of unit.

All of the following are valid ways of resolving the unit inconsistency:

```
inUnit(1.0'cm', "m") + 0.5 + 1.0'm'
inUnit(1.0'cm' + 0.5, "m") + 1.0'm'
1.0'cm' + inUnit(0.5 + 1.0'm', "cm")
1.0'cm' + 0.5 + inUnit(1.0'm', "cm")
```

Note how the interpretation of the literal `0.5` is affected by where `inUnit` is applied. This demonstrates that the explicit use of `inUnit` (or `autoUnit`) gives control over the interpretation of empty unit expressions.

This is also where the danger of implicit unit conversion lies; if the inconsistency was automatically resolved by the tool introducing `inUnit` somewhere in the expression, the interpretation of `0.5` would seem ambiguous, especially to someone who expected addition to be commutative and associative. The authors believe that even if a unique choice of unit for `0.5` could be defined without sacrificing commutativity and associativity, the rules for implicit conversion would become too unintuitive to be of practical relevance with human users in mind. Instead, implicit unit conversion needs to be constrained so that the interpretation of empty unit expressions remains as intuitive as in current Modelica.

In subsection 6.2, the rules for implicit unit conversion will be defined so that the problematic cases above can be rejected.

6.2 Extensions to unit checking machinery

This section describes how the unit checking machinery from section 3 is extended to support implicit unit conversion. The unit semantics resulting from the extended machinery is denoted the *extended semantics*.

It is observed that in almost all situations where a unit constraint is introduced according to the current semantics, at least one side of the constraint corresponds to an expression which could be wrapped in `inUnit(...)` to resolve a unit inconsistency. The two exceptions are `unit`-attributes and `connect`-equations.

The `unit`-attribute exception is not a problem, since there are good reasons anyway to constrain the unit inference by giving these constraints priority over all other constraints in the sense that a unit variable is always solved from its `unit`-attribute constraint in case one exists. It follows that these constraints are never resolved using implicit unit conversion. Besides ensuring that `unit` -attributes are obeyed exactly for the variable they belong to, this also provides an important mechanism for delimiting the potential issues associated with allowing implicit unit conversion.

A `connect`-equation constraint is an exception since neither operand of `connect` is allowed to be an `inUnit(...)` expression, and this exception will be dealt with towards the end of this section.

The extended semantics for the non-exceptional cases is that unit constraints are now introduced in the form of *directed* equivalence compatibility of two unit meta-expressions. When x is required to be equivalence compatible to y, the direction from x to y shall indicate that x corresponds to an expression that can be unit converted if needed. When a unit constraint free of unit variables is obtained during inference, it is first checked whether it is equivalence compatible. If it is, there is no need of unit conversion and the constraint can safely be dropped. Otherwise, if it is not convertibility compatible, there is a unit inconsistency that implicit conversion cannot resolve. In the remaining case, implicit unit conversion is tentatively injected on the source side of the constraint in order to establish equivalence compatibility. After completed unit inference, the tentatively introduced conversions are analyzed to ensure that they do not introduce ambiguous values of quantity. This analysis is denoted *robustness analysis* and will be described below.

Computations where all `Real` expressions have units, and the units are used consistently, have uniquely defined values of quantity as outcome – the alternative would have been a serious flaw of the current semantics. Similarly, computations where all `Real` expressions have empty unit, are interpreted as operations on real numbers, and since no units are present, there is no risk of ambiguity due to automatic unit conversions. The risk of ambiguity arises where the two kinds of computation meet, due to the Modelica convention of treating an empty unit expression as if it had a unit which fulfills a unit constraint with a concrete unit on the other side.

In the extended semantics, units determined by the unit inference machinery are only uniquely determined up to convertibility. For example, a variable could end up getting the unit `"J"` or the unit `"kN.m"` depending on how the unit constraints are processed, but the ambiguity in the unit of a variable isn't by itself causing ambiguous values of quantity. The ambiguous values of quantity arise when an ambiguous unit is used to interpret an empty unit expression as a value of quantity.

Ambiguous units originate where a tentative unit conversion is introduced, since it is generally not uniquely defined where the conversion must be introduced. Such a unit constraint is denoted an *origin of ambiguity*. The ambiguity of the constraint gets propagated to any unit variable solved from the constraint. If a unit variable is found to be ambiguous, the ambiguity gets propagated to all unit constraints where the unit variable is present. If the variable `x` has a `unit`-attribute, then x_u does not propagate ambiguity and is said to be *robust by definition*. If the propagation of ambiguity back and forth between constraints and unit variables reaches a constraint where the empty unit is on one side of the constraint, then an ambiguous value of quantity has been detected, a situation which will be denoted a *robustness conflict*.

Thus, robustness analysis comes down to showing that there are no robustness conflicts. This problem is naturally studied on the bipartite incidence graph of unit constraints and unit variables. A constraint where one side is empty or undefined is said to be an *origin of fragility*. Existence of a robustness conflict is equivalent to existence of a path in the graph from an origin of ambiguity to an origin of fragility, after excluding the variables which are robust by definition. Such a path is denoted a *robustness conflict path*.

Note that the origins of ambiguity in a robustness analysis graph depend on the order in which the unit constraints

DOI
10.3384/ecp21827

are processed during unit inference, so other implementations are likely to produce different graphs. However, if one implementation marks a constraint as an origin of ambiguity, another implementation will also mark a constraint as origin of ambiguity within reach from the former constraint. Hence, the existence of a robustness conflict path will not depend on the order in which the constraints are processed.

Returning to the exception of `connect`-equations, the only difference in the handling compared to other equations is that the introduction of `inUnit(...)` to resolve unit conflicts is delayed until the connection set equations have been generated. For instance, the robustness analysis makes no difference between unit constraints coming from `connect`-equations and unit constraints coming from other equations; the need for implicit unit conversion in the constraint of one `connect`-equation will be an origin of ambiguity, and as all the connectors in the connection set will be connected in the robustness analysis graph, the ambiguity will reach the entire connection set. At the time of elaborating the connection set equations, it must be remembered that the established unit consistency of the `connect`-equations is only for convertibility, not equivalence. For each generated flow or potential equation a common unit for the equation is selected among the ones which are already present; this will ensure that no implicit unit conversion in the connection set equation will take place when all units are equivalent. Then all terms of the flow or potential equation are conceptually converted to the common unit using `inUnit(...)` (which will be a no-op in case the term already has a unit which is equivalent to the common unit).

6.3 Examples

Examples in this section will be illustrated using the following conventions for the display of a robustness analysis graph:

- Constraint vertices are placed to the left, and variable vertices to the right.
- An origin of ambiguity is colored in blue, and an origin of fragility is colored in orange. The remaining constraint vertices have dark gray color.
- A variable which is robust by definition is colored in light gray, while other variables have dark gray color.
- All edges incident to robust variables are colored in light gray (they cannot be part of robustness conflict paths). Edges reachable from an origin of ambiguity are drawn with an arrow in the direction of increasing distance to the origin of ambiguity. The remaining edges are drawn with dashed line and without arrow.
- In a constraint, empty is presented as ∅, and undefined as ?.

With this representation, existence of a robustness conflict path corresponds to an orange vertex with an incoming arrow. A robustness conflict path can then be reconstructed by following the directed edges in reverse direction until a blue vertex is reached. Note that the robustness analysis graphs presented below correspond to one of many possible orders of processing the unit constraints, but that while other implementations are likely to find other origins of ambiguity, the existence of a robustness conflict path does not depend on the order in which the constraints are processed.

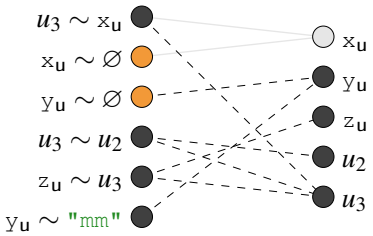

Figure 1. Robustness analysis graph without conflict path.

Consider the following simple model with explicit automatic unit conversion:

```
Real x(unit = "m") = 0.1;
Real y(min = 1.0'mm') = 10.0;
Real z = x + autoUnit(y);
```

The robustness analysis graph in Figure 1 contains origins of fragility, but no origins of ambiguity since all unit constraints are fulfilled without implicit unit conversion. Without any origins of ambiguity there cannot be any robustness conflict.

Now consider the same model again, but without `autoUnit`:

```
Real x(unit = "m") = 0.1;
Real y(min = 1.0'mm') = 10.0;
Real z = x + y;
```

The robustness analysis graph in Figure 2 contains a robustness conflict path leading to the fragile constraint for the binding of `y`.

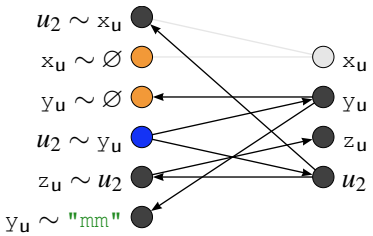

Figure 2. Robustness analysis graph with conflict path.

By adding a `unit`-attribute for `y`, the robustness conflict path is cut by the removal of the node y_u. Then, add the literal `1.0` the sum:

```
Real x(unit = "m") = 0.1;
Real y(unit = "mm") = 10.0;
Real z = x + 1.0 + y;
```

The robustness analysis graph in Figure 3 contains a robustness conflict path also this time.

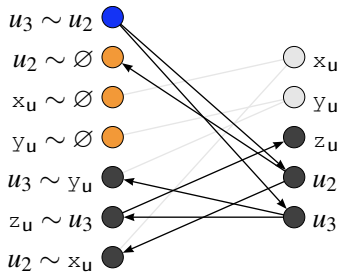

Figure 3. Robustness analysis graph with conflict path.

Now change units so that unit conversion is not needed:

```
Real x(unit = "m") = 0.1;
Real y(unit = "m") = 0.01;
Real z = x + 1.0 + y;
```

This is robustness analysis applied to existing Modelica models in a nutshell; there are plenty of origins of fragility, but no origins of ambiguity. As expected, the robustness analysis graph in Figure 4 has no robustness conflict path.

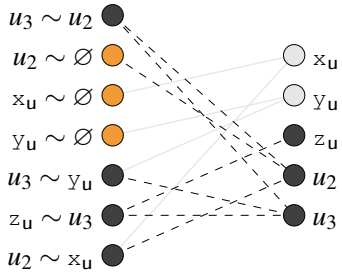

Figure 4. Robustness analysis graph without conflict path.

Robustness analysis doesn't require that all unit variables have been determined:

```
Real x = 2.0;          // Undetermined unit
Real y = 3.0;          // Undetermined unit
Real z = 1'm' + 100'cm';
Real u = x + y * z;
```

As expected, the robustness analysis graph in Figure 5 has a robustness conflict path.

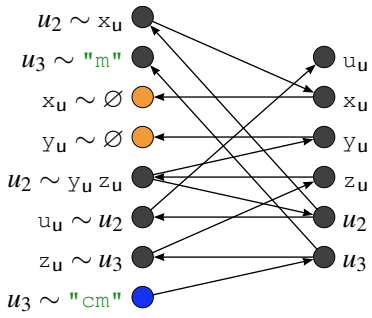

Figure 5. Robustness analysis graph with conflict path.

A consequence of the desire to avoid the "wildcard effect" of empty literals in current Modelica is that multiplication does not distribute over addition when empty literals are present:

```
Real x(unit = "m") = 2.0;
Real y(unit = "m") = 3.0;
Real u = x * (1.0 + y);    // 6.0'm2'
Real v = x * 1.0 + x * y; // Inconsistent
```

Since there is a constraint which cannot be resolved using implicit unit conversion, robustness analysis is neither needed nor applicable.

The following equation is considered consistent in units according to current Modelica, and is therefore also considered valid with the extended semantics:

```
  Real x(unit = "m");
equation
  x = 2.0;
```

Note how the literal `2.0` gets its unit from the other side of the equation. The robustness analysis graph in Figure 6 has no robustness conflict path.

Figure 6. Robustness analysis graph without conflict path.

Now try to adding the same value of quantity to both sides of the equation:

```
  Real x(unit = "m");
equation
  x + 300.0'mm' = 2.0 + 300.0'mm';
```

Looking at the right-hand side alone, the addition of 300 millimeters has changed it from 2000 millimeters to 302 millimeters, a quite unpleasant surprise. It is reassuring to see that the robustness analysis graph in Figure 7 has a robustness conflict path.

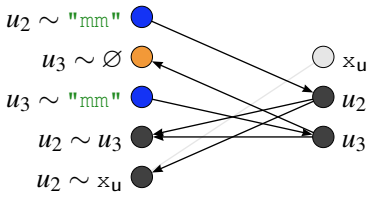

Figure 7. Robustness analysis graph with conflict path.

Consider subtracting the same literal without unit from both sides of an equation:

```
  Real x1(unit = "m");
  Real x2(unit = "m");
equation
  x1 + 1 = 1.0'mm';
  x2 = 1.0'mm' - 1;
```

The solutions to the two equations are not the same, that is, the solutions for `x1` and `x2` are poorly defined values of quantity. Again, it is reassuring to see that the robustness analysis graph in Figure 8 has two independent robustness conflict paths.

DOI 10.3384/ecp21827

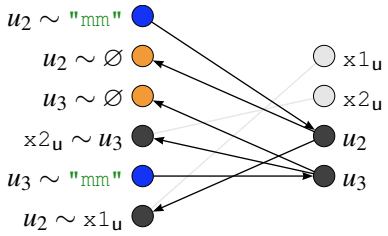

Figure 8. Robustness analysis graph with two independent conflict paths.

Ambiguity can spread through multiplicative constraints:

```
  Real x = 1'cm' + 1'm';
  Real y;
  Real z = y + 1.0;
equation
  1.0'J' = x * y;
```

The robustness analysis graph – omitted due to space constraints – contains a robustness conflict path via y_u.

Variation of the previous example:

```
  Real x = 1'cm' + 1'm';
  Real y;
equation
  1.0'J' = x * (y + 1.0);
```

The robustness analysis graph in Figure 9 has a robustness conflict path. That giving `y` a `unit`-attribute does not help is a sign of the conservativeness of the proposed semantics.

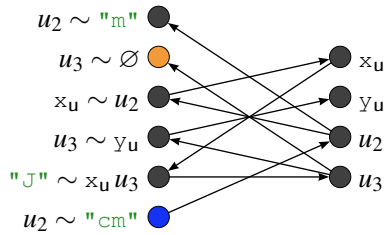

Figure 9. Robustness analysis graph with conflict path.

The trivial resolution of the conflict is to attach the intended unit to the literal:

```
  1.0'J' = x * (y + 1.0'kN');
```

When units are undefined, the possibility of implicit unit conversion must not invalidate models that are valid according to current conventions:

```
function f
  input Real u;
  output Real y = u;
end f;

Real x(unit = "m") = 2.0;
Real y(unit = "mm") = 300.0;
Real u = y + f(x);
Real v = u + 1.0;
```

Intuitively, however, `f(x)` should have had unit `"m"`, and there should be a need for unit conversion resulting in an origin of ambiguity incident to u_u, which in turn would reach the origin of fragility from `u + 1.0`. What actually happens is that `f(x)` has undefined unit, meaning that there is no origin of ambiguity in the model and that the model is accepted also according to the extended semantics.

Now take a model which is inconsistent according to current Modelica:

```
Real u = 1.0'cm' + f(2.0'm');
Real v = 1.0'cm' + u + 1.0'mm';
```

The function call's undefined unit will lead to an origin of fragility, and the need for implicit unit conversion means that there will be at least one origin of ambiguity in the robustness analysis graph. Without any robust variables in the graph, it is easy to see that a robustness conflict path exists. This is in agreement with what would happen if `f(2.0'm')` was inlined, so that there would be no undefined units in the model.

While robustness analysis of equation-based models is essential for illustrating the principles, the final example in this section is a component-based model where implicit conversion is needed in a connection set. Unlike some of the nontrivial examples above, it is representative of the simple way in which implicit conversion matters most to applications. That is, `unit`-attributes are present near the place of implicit conversion on both sides, shielding any origins of fragility in other parts of the model from being reachable from the origin of ambiguity:

```
  Sources.Constant a(k(unit = "m/h") = 1);
  Math.Gain b(u(unit = "mm/s"), k = 1);
equation
  connect(a.y, b.u);
```

7 Applications

In the following, let `Convert` be the block version of the `inUnit` operator, that is, a block for explicit conversion to a given unit. Similarly, let `Value`, `NumericValue` and `AutoConvert` be the block versions of the `withUnit`, `withoutUnit` and `autoUnit` operators.

7.1 Eliminating conversion blocks

In current Modelica, a unit conversion block can be created by careful parameterization of a gain block:

```
final Gain from_MW_to_W(
  u(unit = "MW"),
  y(unit = "W"),
  k(unit = "W/MW", displayUnit = "1") = 1e6
);
```

While the use of `displayUnit = "1"` helps entering and verifying that the value of `k` is correct for the current choice of source and target units (compare subsection 1.3), the setup is still error-prone and users who are not familiar with gain blocks used to perform unit conversion could find the use confusing.

With `inUnit` or `autoUnit`, defining a reusable unit conversion block is straight-forward, but unnecessary

when implicit unit conversion is allowed. Instead of using a unit conversion block between a source with unit `"MW"` and a sink with unit `"W"`, the source and sink may be connected directly. While this may appear strange with the unit equivalence mindset, it is the natural way of connecting the blocks with the value of quantity mindset.

Sometimes, the shift of mindset alone may be enough to eliminate a conversion block. Here, if the sink would have been an integrator for computing total amount of work based on input power, the value of quantity mindset makes it less important whether the unit of the total work is `"W.s"` or `"MW.s"`. For instance, a user might prefer to use a display unit such as `"J"` or `"kW.h"`, and then the underlying unit will not matter for the presentation.

7.2 Working with table data

When table data is expressed in different units than what is used in the rest of a model, there are several interesting options based on the proposed extended semantics. As before, the starting point is to ensure that the table component for accessing the data has connectors with correctly specified `unit`-attributes.

By putting a `Convert` block after a table output, it is possible to read table data in an explicit choice of unit. In a model made without declared units, a `NumericValue` block can be used instead of the `Convert` block to access the table data in the undeclared units assumed by the model. In a model with declared units, a `Convert` block will generally be redundant in view of implicit unit conversion, but if there is a preference of unit conversion to happen as close as possible to the table block, an `AutoConvert` block can be used to pinpoint the transition between the table's units and the units used in the rest of the model.

Similar options exist handling the need for unit conversion of table inputs.

7.3 FMI import

Although the creation of an FMU wrapper model for each imported FMU opens up the possibility of adding unit conversion blocks in the wrapper model, the recommended approach is to use FMU's units in the interface of the wrapper model. With this approach, working with FMUs is even easier than working with table data, since the FMI standard makes it a responsibility of the FMU import mechanism to set up correct units in the wrapper model interface.

7.4 Connecting with external processes

The typical way of connecting a Modelica model to an external process is to use a library providing blocks for reading and writing data over some process communication protocol. Again, the important part of the setup is to ensure that the block instances for reading and writing data have correctly configured `unit`-attributes. When supported by the communication protocol, the library should verify that the units configured for the read blocks agree with incoming data, and that the units configured for write blocks are transmitted with the written data.

7.5 Combination with unitful literals

Consider setting a length parameter to 40 feet. In current Modelica, it is not easy for a human looking at the source code to see that the parameter has a simple value in the `displayUnit`:

```
parameter Real p
  (unit="m", displayUnit="ft") = 12.192;
```

By combining constructs proposed in this work, the intent of the parameter setting can be made clear also to humans looking at the source code by writing `40'ft'` directly in the declaration equation. Further, if the preferred unit for plotting the result is not feet, the `displayUnit` can be changed independently of the unit used to set the parameter value.

It is observed that adding dedicated graphical user interface support for this way of expressing declaration equations would provide a clean solution to the problem of how many digits to present when display unit conversion is lossy due to finite precision, such as in the common case of having degrees as display unit for angles in radians.

7.6 Comparing a ratio to a threshold

Consider some variants of (attempting to) compare a ratio to a threshold:

```
Real x = 1.0'm';
Real y = 1.0'cm';
Boolean b1 = x / y > 50;
Boolean b2 = inUnit(x / y, "1") > 50;
Boolean b3 = x / y > 50'1';
```

In the first case (`b1`) one gets the quite surprising result that one meter is not more than 50 times longer than one centimeter. The second variant is the most verbose, but gives the desired result and does not rely on implicit conversion. The third variant is the most compact thanks to relying on implicit conversion.

8 Future work

It might come as a surprise that the proposed extended semantics considers the following a robustness conflict:

```
Real x1 = 1.0'm' + (1.0'mm' + 1.0);
```

That is, not even parentheses can be used to remove ambiguity. One idea for future work is therefore to make the robustness analysis aware of the parentheses.

Depending on how the methods proposed here for locally taking control over the implicit unit conversions are received, there could be reasons to explore other options as well. This would include the `FixedUnits` approach taken by Unitful.jl, but the Modelica setting also offers the possibility of specifying an annotation on the level of a class or equation for disallowing implicit unit conversion.

The relation between evaluable variability and unit conversion needs clarification. Here, more work is required

DOI
10.3384/ecp21827

to analyze the tradeoff between complexity of interleaving parameter evaluation and unit inference, and the limitations that would come with more restrictive approaches with less implementation effort.

There are currently several tool-specific mechanisms for defining additional unit symbols beyond the ones required by the Modelica specification, but a standardization effort is needed to make use of these units portable across tools. Alternatively or as a complement, the need for customization could easily be mitigated by adding many more well established units to the Modelica specification, and having well established units in the specification would also reduce risk of conflicting custom definitions or multiple symbols being introduced for the same unit.

When understanding of unit checking for units without offset in Modelica has matured, the time would be ripe for a discussion about units with offset, that is, the topic of quantity space checking. With quantity space checking, it becomes possible to detect an inconsistency when computing the difference between two absolute temperatures in degree Celsius and kelvin. When combined with implicit unit conversion, the inconsistency is automatically resolved by converting one operand to the unit of the other before carrying out the subtraction.

In the long run, it would be interesting to pursue a development in another direction, namely to support moving the Modelica ecosystem away from the legacy of allowing expressions with empty unit to be interpreted as values of quantity based on units taken from the context. This would align unit handling in Modelica with unit handling in other popular languages, and could eventually completely eliminate the need for the robustness analysis presented in this work. By only keeping the possibility of interpreting empty unit expressions as values of quantity with unit `"1"`, the threshold comparison in subsection 7.6 could finally be correctly expressed simply as `x / y > 50`.

9 Summary and conclusions

It has been demonstrated how implicit unit conversion can be safely allowed in Modelica. The crucial difference to other languages and software for unit aware computation is the Modelica legacy of allowing expressions without unit to be interpreted as values of quantity based on units taken from the context. When implicit unit conversion is allowed, this creates a risk of interpreting such expressions based on units which are not uniquely determined. A simple and conservative condition for rejecting programs in risk of ambiguously determined values of quantity has been presented. The condition will allow implicit conversion in typical situations where it is useful, as well as never reject any Modelica program which is valid according to current unit semantics. In addition to implicit unit conversion, a set of basic operators for working with units and values of quantity has been proposed. These operators add value on their own, and can also be used to locally take control over the implicit unit conversions.

The importance of understanding a model in terms of related values of quantity rather than merely seeing units attached to variables as decorative ornaments has been stressed. Without this understanding, implicit unit conversion in Modelica would make as little sense as it would in any of the other languages and softwares that allow implicit unit conversion today.

The usefulness of the proposed unit semantics has been demonstrated in several real world examples of Modelica use, and the safety of the approach has been illustrated with numerous minimal test cases.

Acknowledgements

This work has been supported by Wolfram MathCore.

References

Aronsson, Peter and David Broman (2009-09). “Extendable Physical Unit Checking with Understandable Error Reporting”. In: *Proceedings of the 7th International Modelica Conference*, pp. 890–897. DOI: 10.3384/ecp09430027.

Hindley, J. Roger (1969). “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146, pp. 29–60. DOI: 10.2307/1995158.

Kennedy, Andrew J. (1997-01). “Relational Parametricity and Units of Measure”. In: *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*. ACM.

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

Maple – Tutorials Part 10: Units (2025). URL: https://www.maplesoft.com/support/help/Maple/view.aspx?path=MaplePortal/Tutorial10 (visited on 2025-04-24).

Math.js – Units (2025). URL: https://mathjs.org/docs/datatypes/units.html (visited on 2025-07-27).

Mattsson, Sven Erik and Hilding Elmqvist (2008-03). “Unit Checking and Quantity Conservation”. In: *Proceedings of the 6th International Modelica Conference*. Vol. 1, pp. 13–20. URL: https://modelica.org/events/conference2008/proceedings/volume_1.pdf.

Milner, Robin (1978). “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17.3, pp. 348–374. DOI: 10.1016/0022-0000(78)90014-4.

Pint – Tutorial (2025). URL: https://pint.readthedocs.io/en/stable/getting/tutorial.html (visited on 2025-04-24).

Simulink – Converting Units (2025). URL: https://se.mathworks.com/help/simulink/ug/convert-units.html (visited on 2025-04-24).

Unitful.jl – Conversion/promotion (2025). URL: https://painterqubits.github.io/Unitful.jl/stable/conversion/ (visited on 2025-04-24).

Wolfram Language – Quantity (2025). URL: https://reference.wolfram.com/language/ref/Quantity.html (visited on 2025-04-24).