Balanced Models in Modelica 3.0 for Increased Model Quality

Hans Olsson[1] Martin Otter[2] Sven Erik Mattsson[1] Hilding Elmqvist[1]
[1]Dynasim AB, Ideon Science Park, SE-223 70 Lund, Sweden
[2]German Aerospace Center (DLR), Institute of Robotics and Mechatronics, Oberpfaffenhofen, 82234 Weßling, Germany
{Hans.Olsson, SvenErik.Mattsson, Hilding.Elmqvist}@3ds.com, Martin.Otter@dlr.de

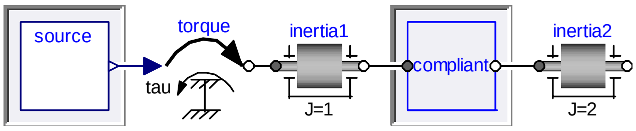
Abstract

A Modelica model can only be simulated, if the number of unknowns and the number of equations are equal. In Modelica 3.0, restrictions have been introduced into the language, in order that every model must be "locally balanced", which means that the number of unknowns and equations must match on every hierarchical level. It is then sufficient to check every model locally only once, e.g., all models in a library. Using these models (instantiating and connecting them, redeclaring replaceable models etc.) will then lead to a model where the total number of unknowns and equations are equal. Besides this strong guarantee, it is possible to precisely pinpoint which submodels have too many equations or lack equations in case of error. This paper gives the rationale behind the Modelica 3.0 design choices including proofs of the new guarantees, and discusses the limitations of this approach.

1 Background

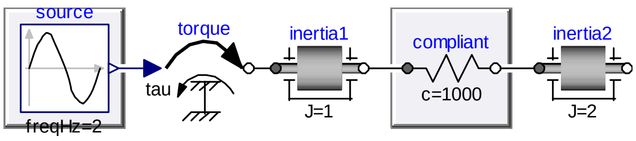
In a causal modeling paradigm, where only input/output blocks are used, it is straightforward to verify that all input connectors have been connected, and thus causal modeling naturally lead to a simple plug and play metaphor for end-users. The goal is to ensure that acausal Modelica model components are as convenient to use for end-users.

The need for this is growing in importance with larger and more complex model libraries and with companies expanding Modelica usage from research to development. Furthermore, libraries with template models, i.e., incomplete models with replaceable components, like VehicleInterfaces, PowerTrain, VehicleDynamics library, can easily lead to wrong models in Modelica 2 when using the templates, without being able to give reasonable diagnostics for the source of the error. Shortened production cycles imply that we want to verify correctness early, in particular already for incomplete models where implementation of the parts is left open. An example below (sunken icons means partial replaceable components) shows a driveline where the input torque and compliance between the inertias are unspecified.



The goal is that by separately verifying that the template model is correct, and imposing restrictions on the models we plug in, we can be certain that the complete model is correct.

Without the restrictions, the tool would need to perform a global analysis, and if the complete model is not balanced we would not know whether the implementation of the part or the template model itself was in error. Having to verify this for all combinations of sources and gears (one is shown below) is not practical:



The first possibility would be to improve the analysis of structural singularities for Modelica 2 to find the errors without requiring balancing, but instead use other information including annotations and confidence in different equations [2]. Similar techniques are also useful at the lowest level to go from one equation too many in the current model to pinpointing which equation is superfluous.

Previously there have been checks in Dymola [3] (since Dymola 5.3 released in 2003) to determine in case of an unbalanced simulation model which submodels are incorrect based on the actual use. That was introduced to help users in finding errors, but it did not always work satisfactorily since it was not always possible to determine how many equations should be present in each model (the best what could be done in general was to determine a range for the number of equations); thus for the complete model it was not possible to determine whether the error was in the template or in one of the implementations – and if so which one. Thus without stricter language rules no reliable diagnostics could be given to users to pin-point the errors.

A related work [4] allows unbalanced classes, maintains the (un)balancing of connectors and models when modified. The difference is that in Modelica 3.0, it is enforced that models are balanced from the start and on all levels.

2 Number of equations in the model

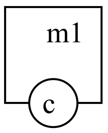
In order to verify whether a component model is balanced we must have a clear definition of how many equations the model contains – and how many equations it should contain (both based on its interface).

2.1 Restrictions on physical connectors

The goal is that combining models having (physical) connectors and connecting them (in legal ways) we should get new balanced models; without imposing additional restrictions on the models or requiring adding equations.

Consider the simplest case of a model where the only public part is one connector, and it is "physical", i.e. containing n_f flow variables "`Real f[nf]`" and n_p non-causal, non-flow, "potential" variables "`Real p[np]`" (i.e., no connector variable has the input or output prefix); and the model is balanced, i.e., the model requires that externally a specific number of equations (n_e) is provided, in order that all unknowns of the model can be uniquely computed together with the internal equations in the model. We will call these required equations in the sequel external equations of a model component. The number of external equations has to be *uniquely* defined by the interface of the model. The balancing is that the number of unknown variables equals the number of equations defined inside the model plus the number of external equations.

The simplest use of a model `m1` with connector `c` is that the connector is unconnected.



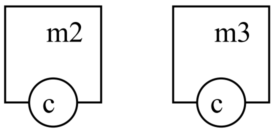
The Modelica semantics does state that the flow-variables are summed to zero, whereas the "potential" variables should be equal [1].

This leads to the external equations

```
m1.c.f = 0; // nf equations
```

Since `m1` requires n_e external equations and instantiating the component gives n_f equations, we have the requirement: $n_e = n_f$.

The next case is to have two of such components not being connected:

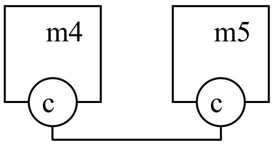


In this case we get the external equations

```
m2.c.f = 0; // nf equations
m3.c.f = 0; // nf equations
```

Since `m2` requires n_e equations and `m3` requires n_e equations, we have the requirement: $2 \cdot n_e = 2 \cdot n_f$

The next case is to have two of such components, but being connected:

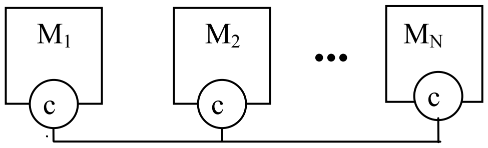


Here we get connection equations:

```
m4.c.f + m5.c.f = 0; // nf equations
m4.c.p = m5.c.p;     // np equations
```

Since `m4` requires n_e equations and `m5` requires n_e equations, we have the requirement: $2 \cdot n_e = n_f + n_p$

The final case is to have N components that are connected together:



Here we get one time the zero-sum equations for the flow variables and N-1 identity equations for the potential variables. Since every model requires n_e equations, we have the requirement:

$$N \cdot n_e = n_f + (N-1) \cdot n_p$$

To summarize, we get the following relations that all have to be fulfilled, in order that instantiating and optionally connecting components does not require to add any more equations (= necessary and sufficient conditions):

1st model (m1):	$n_e = n_f$
2nd model (m2,m3):	$2n_e = 2n_f$
3rd model (m4, m5)	$2n_e = n_f + n_p$
4th model (M_1, ...M_n)	$N \cdot n_e = n_f + (N-1) \cdot n_p$

or equivalently

$$n_f = n_e$$

$$n_p = n_f$$

This leads to the conclusion that the number of flow and non-causal, non-flow variables must match (counting arrays as the number of elements of simple types), and this must correspond to the number of external equations for this connector.

Models may also have variables that are declared with the input prefix, both in a declaration of a model and in a (top-level) connector. These variables are treated as unknowns in a model. It is natural to require that for all these input variables external equations must be provided.

In order to force the user that all missing equations for a model are provided when instantiating the model, it is required that all input variables declared in a model are provided as modifiers and that all inputs in a (top level) connector are provided by connecting the connector. Since a connector must have the same number of flow and potential variables (see derivation above), the means that a connector with an input variable A must be connected to another connector where variable A has the output prefix (an exception of this last rule will be discussed in the next section).

According to these rules, it is no longer allowed to provide modifiers for other variables (with exception of variables declared with the input, parameter or constant prefix) or add other equations for the component externally, because all of these actions would introduce superfluous equations.

To summarize, we have basically the following requirements (for simplicity, not yet considering special cases such as over-determined connectors, non-causal variables with a declaration equation, partial models, or connectors with input variables that are not connected):

1. The number of flow variables in a connector must be identical to the number of non-causal, non-flow variables (variables that do not have a `flow, input, output, parameter, constant` prefix).
2. The number of equations in a model = number of unknowns – number of inputs – number of flow variables (of top-level public connector components). For the equation count, components are not taken into account, because this is taken into account by the next rule 3.
3. When using a model, i.e., making an instance, all missing equations of this component must be provided to make the component "balanced" by:
   a) Connecting connectors or by leaving "physical" connectors unconnected (since the missing equations are then automatically introduced by setting all flow variables to zero).
   b) Providing a modifier for every non-connector component variable with an `input` prefix. Besides parameters and constants, modifiers on other variables are no longer allowed.

The above rules shall be clarified with a few simple examples (assuming a global definition `import SI=Modelica.SIunits;`):

```
connector FluidPortA
  SI.Pressure                p;
  flow   SI.MassFlowRate     m_flow;
  input  SI.SpecificEnthalpy h_inflow;
  output SI.SpecificEnthalpy h_outflow
end FluidPortA;
```

```
connector FluidPortB
  SI.Pressure                p;
  flow   MassFlowRate        m_flow;
  output SI.SpecificEnthalpy h_inflow;
  input  SI.SpecificEnthalpy h_outflow
end FluidPortB;
```

The two connectors FluidPortA and FluidPortB are valid, since they each have 1 flow and 1 non-causal, non-flow variable and 2 causal variables. Note, whenever input/output prefixes are present, there are connection restrictions because the block diagram semantics holds (e.g. an output cannot be connected to an output). As a result FluidPortA can only be connected to one FluidPortB, but not to another FluidPortA.

```
connector WrongFlange // wrong connector
   SI.Angle            angle;
   SI.AngularVelocity  speed;
   flow SI.Torque      torque;
end WrongFlange;
```

This connector is not valid, since the number of flow and non-flow variables is not the same. This is a typical situation of "old" connectors, such as the connectors of the (obsolete) ModelicaAdditions.MultiBody library. Both these "old" connectors, as well as the "WrongFlange" connector above can be made valid, by using the prefix input or output for one of the non-flow variables (similarly to FluidPortA and FluidPortB above).

```
model Pin
   SI.Voltage     v;
   flow SI.Current i;
end Pin;
```

```
model Capacitor
   parameter SI.Capacitance C;
   SI.Voltage u;
   Pin p, n;
equation
   0 = p.i + n.i;
   u = p.v - n.v;
```

```
     C*der(u) = p.i;
  end Capacitor;
```

The Capacitor model has 5 unknowns[1] (`u, p.v, p.i, n.v, n.i`) and 2 flow variables (`p.i, n.i`). It is therefore required that this model has 5 – 2 = 3 equations, and the model fulfills this requirement.

```
  model Test1
    Capacitor C1(C=1e-6);      // o.k
    Capacitor C2(u=sin(time)); // wrong
  end Test1;
```

The declaration of C1 is correct, because a modifier for a parameter is given. The declaration of C2 is wrong, because it is no longer allowed in Modelica 3 to provide a modifier for a variable that does not have a `constant`, `parameter` or `input` prefix.

```
  model VoltageSource
     input SI.Voltage u;
     Pin p, n;
  equation
     u = p.v - n.v;
     0 = p.i + n.i;
  end VoltageSource;
```

The VoltageSource model has 5 unknowns (`u, p.v, p.i, n.v, n.i`), 2 flow variables (`p.i, n.i`) and 1 input variable (`u`). It is therefore required that this model has 5 – 2 – 1 = 2 equations and the model fulfills this requirement.

```
  model Test2
    ...
    VoltageSource V1(u=sin(time)); // o.k
    VoltageSource V2;              // wrong
    ...
  end Test2;
```

Component V1 is correct, because the missing external equation for the unknown input `u` is given as modifier.

Component V2 is not correct, because no modifier or equation is provided for the missing unknown input "`u`".

The counting for non-connector inputs (such as u) is defined as if they always had a declaration equation. Thus the result would be the same for this modified model:

```
  model VoltageSource
     input SI.Voltage u=0; // Default
     Pin p, n;
  equation
     u = p.v - n.v;
     0 = p.i + n.i;
  end VoltageSource;
```

[1] Alternatively, one could define „u" as known (because it is a potential state) and „**der**(u)" as unknown. However, this does not hold in general, since **der**(..) might have an expression as argument. For this reason, **der**(..) is used as operator that does not have an influence on the equation counting.

This implies that we can add default values without modifying the use of the model.

2.2 Correlations and non-connector inputs

Causal variables are not limited to connectors, but there are also non-connector inputs and outputs – which can be viewed as "time-dependent parameters". The non-connector outputs have no special significance here (they are useful to indicate special interesting variables). The non-connector inputs are for the balancing always counted as having a binding equation – and must have a binding equation in the complete model. This simplifies the requirement for counting equations such that modifiers are not counted as providing external equations for the model; since they are seen as replacing old declaration equations with new ones of similar size. The important aspect is that other alternatives, such as giving normal equations for them – or modifying some non-input would not preserve the balancing of equations.

One example demonstrating this issue is correlations; i.e., relations constraining a set of variables to be on a hyper-plane of a certain dimension. The simplest is a correlation involving two variables; in this case the variables will simple be on a curve. We can arbitrarily declare one as input, but the correlation normally is written as just an equation relating the variables (and the line could have straight segments in both x and y direction). *Note: This example is illegal in Modelica 3.0.*

```
partial model Correlation
  input Real x;
        Real y;
end Correlation;

model UseCorrelation
  // Wrong in Modelica 3
  replaceable Correlation corr;
equation
   corr.y=2+time;
   /* Same number of equations as
      modifying "x"; could also be
      written as modifier for y */
end UseCorrelation;

model LineCorrelation
  extends Correlation(x=3);
equation
   x+y=0;
end LineCorrelation;

model Complete=UseCorrelation
     (redeclare LineCorrelation corr);
// model is not balanced since
// 2 unknowns (x,y), but 3 equations:
//   x + y = 0;
//   x = 3;
//   y = 2 + time;
```

In practice we would have a set of correlations and a set of uses of them, and what we want to verify is that they are all correct without performing all the tests (which would lead to a combinatorial explosion in the number of tests). For Modelica 3.0 we wanted to support the model `Correlation` and the use in `LineCorrelation` and `Complete` – without the possibility of too many equations in the `Complete` model. Since we would expect these models to be developed by separate teams (and normally be part of larger systems) it represents exactly the situation to avoid – an unbalancing due to the interaction of several correct models – and without a clear description of where the error is. The solution is in this case to disallow the construct in `UseCorrelation` for non-connector inputs, and find another way of providing the correlations: Component "`corr`" in `UseCorrelation` has one input variable and it is required to provide a modifier for this variable in order to make model "`corr`" balanced. This is not the case above and therefore the model is not correct. This restriction on modifiers to parameters, and non-connector inputs is part of Modelica 3.0.

The ideal solution for modeling the correlation would be that `UseCorrelation`, `LineCorrelation` (except for '`x=3`'), and `Complete` would all be legal, and rewrite `Correlation` to allow this. The first attempt was to have some way to disable the balancing test for `Correlation` and derived classes; that implied that only the class `Complete` could be checked; and in case it failed it would be impossible to determine whether `UseCorrelation` or `LineCorrelation` should be modified. During the design of Modelica 3.0 several attempts were made of introducing a special syntax for stating that `Correlation` is lacking a certain number of equations – without defining 'x' as an input (because either x or y shall be defined when using the component). This requires the introduction of additional non-intuitive syntax and the final decision was to change instead 'x' to a connector input and modify the language rules to allow unconnected connector inputs and provide the binding equation for the input connector as equation. The example then becomes (the differences in Correlation are highlighted):

```
partial model Correlation
  InputReal x;
       Real y;
  connector InputReal=input Real;
end Correlation;

model UseCorrelation
  replaceable Correlation corr;
equation
   corr.y=2+time;
end UseCorrelation;
```

```
model LineCorrelation
  extends Correlation;
equation
   x+y=0;
end LineCorrelation;

model Complete=UseCorrelation
     (redeclare LineCorrelation corr);
```

In this case `LineCorrelation` may not use modifiers for '`x`', since '`x`' is a connector, and we are thus once more certain that the number of equations will automatically balance. This is used for Modelica.Media, and can be used in other cases for correlations as well.

Note that UseCorrelation is exactly identical to the original version, but is now legal due to the change in the Correlation model (i.e. `corr.y = 2 + time`, is the missing equation for the input connector `x`).

This approach was decided upon even though it has the disadvantage that we allow unconnected input connectors, and to count equations we thus have to combine the normal equations and the equations for missing input connections in the count of equation. In this case '`corr.x`' is not connected in the `UseCorrelation` model; and instead a non-connect equation is giving.

This can be compared to a causal paradigm, where we would just require that '`corr.x`' must be connected. However, even if the use above is legal a tool could still inform the user that '`corr.x`' lacks a connection if `UseCorrelation` or `LineCorrelation` are not balanced or if the simulation model is structurally singular, in order to help in pinpointing the error.

3 Locally balanced models

In the previous chapter, the counting rules have been sketched for the most important cases. We will now formulate the exact rules and what guarantee can be given:

A model or block is called "locally balanced" if the local number of unknowns matches the local equation size (both terms are defined below). Note, that all counts are performed after expanding all records and arrays to a set of scalars of primitive types. We will here ignore inner and outer components, as well as over-determined connectors, to simplify the definitions and results – for complete definitions see the Modelica 3.0 specification [1].

The local number of unknowns is the sum of:

- For each declared component of specialized class type (Real, Integer, String, Boolean, enumeration and arrays of those, etc) or record it is the "number of unknown variables" inside it (i.e., excluding parameters and constants).
- For each declared connector component, it is the "number of unknown variables" inside it (i.e., excluding parameters and constants).
- For each declared block or model component, it is the "sum of the number of inputs and flow variables" in the (top level) public connector components of these components.

The local equation size is the sum of:

- The number of equations defined locally (i.e. not in any model or block component), including modifier equations, and equations generated from connect-equations.
- The number of input and flow-variables present in each (top-level) public connector component, i.e. the externally needed equations.
- The number of (top level) public input variables that neither are connectors nor have binding equations, i.e., further externally needed equations.

The following restrictions are imposed in Modelica 3.0

- **All non-partial model and block classes must be locally balanced.**
- In a non-partial model or block, all non-connector inputs of model or block components must have binding equations (i.e. they are defined in a modifier).
- Modifiers for components shall only contain redeclarations of replaceable elements and binding equations for parameters, constants (that do not yet have binding equations), inputs and variables having a default binding equation.
- In a connect-equation the primitive components of the two connectors must have the same primitive types, and flow-variables may only connect to other flow-variables, causal variables (input/output) only to causal variables (input/output).
- A connection set of causal variables (input/output) may at most contain one inside output connector or one public outside input connector. [i.e., a connection set may at most contain one source of a signal, which is the "usual" semantics for block diagrams.]
- At least one of the following must hold for a connection set containing causal variables:

  (1) the model or block is partial,

  (2) the connection set includes variables from an outside public expandable connector,

  (3) the set contains protected outside connectors,

  (4) it contains one inside output connector, or

  (5) one public outside input connector, or

  (6) the set is comprised solely of an inside input connector that is not part of an expandable connector.

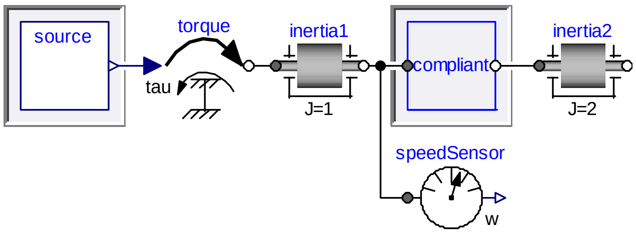
  i.e., a connection set must – unless the model or block is partial – contain one source of a signal (the last items covers the case where the input connector of the block is unconnected and the source is given as equation in the equation or algorithm section).
- A protected outside connector must be part of a connection set containing at least one inside connector or one declared public outside connector (i.e., it may not be an implicitly defined part of an expandable connector).

4 Plug and play

We will show that if a user uses locally balanced classes and follow the language restrictions and drags and drop components and connect them, they will automatically build locally balanced classes as shown below. We will go through this starting from an even more restricted case; in the conclusion we will explain why these rules are not present in the language.

4.1 Only components and connections

Assume we build a non-partial model (or block) composed solely of components of model and block classes (with optional legal value modifiers applied) and connections that satisfy all restrictions, as it is the case in the following figure:



Furthermore for connection sets involving causal variables the connection set should satisfy case 4 in the itemized lists above (=contain an inside output connector generating the signal) – i.e. explicitly excluding case 6 (since cases 1, 2, 3, and 5 cannot apply here). The model or block is then automatically locally balanced.

Note: The excluded case (6) would correspond to removing the source-component above, and instead write a textual equation for torque.tau. This also applies to case (3), which is less needed.

In this case the local number of unknowns corresponds to the number of inputs and flow variables in the public connectors of the components; and the equations to the equations generated by connection equations.

We can split the connectors into causal and non-causal parts (due to the restriction that connection sets may not mix the two; this restriction was added in Modelica 3.0 to allow this analysis).

For the causal part we have the local number of unknowns corresponding to the number of inputs in the public connectors of the components. Among these variables we have n_i inputs and n_o outputs, and the number of equations is thus $n_i + n_o - 1$; the case 4 above gives $n_o = 1$ yielding the local number of equations $n_i + n_o - 1 = n_i + 1 - 1 = n_i$ exactly matching the local number of unknowns.

For the non-causal part we have the local number of unknowns corresponding to the number of flow variables in the public connectors. Assume there are n connectors in this set, and each connector has n_f flow-variables and $n_p = n_f$ non-causal, non-flow variables ("potential variables"). We have $n \cdot n_f$ local number of unknowns; one zero-sum equation for flow variables and $n - 1$ equality equations for the potential variables; in total this gives the size

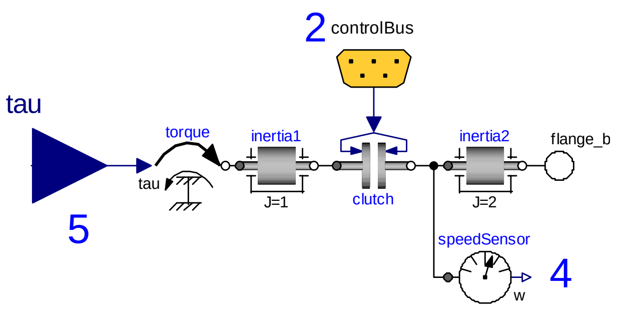
$$n_f + (n-1) \cdot n_p = n_f + (n-1) \cdot n_f = n \cdot n_f$$

which exactly matches the local number of unknowns.

This case is important for users combining models from different libraries – and ensures that as long as the user only combines correct models without introducing simple variables or equations, the model is automatically balanced.

4.2 Connectors

Assume we extend the above list to include components of connectors classes (without any value modifiers), that cases 3 and 6 in the itemized list above does not apply, and that each connector component is connected (case 1 does still not apply since we build a non-partial model or block). The cases are indicated below.



We will apply the split into causal and non-causal part. For the non-causal part it seems that the previous proof goes through automatically – this is true with one minor caveat: if a (public) outside connector had not been connected (the case we excluded) it would *not* have been part of a connection set and would have given 0 equations instead of the correct number n_f.

For the causal part it is more complex since we have both protected and public causal variables. If we disregard case expandable connectors and use superscript i/o for inside/outside, and subscript n for nodes (regardless of input/output).

The unknowns are given by the local connectors: n_i^o outside inputs, n_o^o outside outputs, n_n nodes (protected connectors), and the subcomponents: n_i^i inside inputs. The number of equations (n-1) is extended with n_i^o outside inputs. Thus for balancing we get the requirement:

$$n_i^o + n_o^o + n_n + n_i^i = (n_n + n_i^o + n_o^o + n_i^i + n_o^i - 1) + n_i^o$$

Simple cancellations gives: $n_i^o + n_o^i = 1$, or stated differently: either case (5) an outside input connector, or case (4) an inside output connector. The either is due to the restriction about multiple sources in a connection set. For expandable connectors (case 2) the same rules apply after we have deduced the causality; this will also influence the number of unknowns.

4.3 Redeclare of components

When redeclaring a component, the missing equations for the component must be either provided via modifier equations (parameters, inputs) or connectors must be connected. When these restrictions are fulfilled, the redeclared component is automatically locally balanced.

One situation has to be treated specially: If the redeclared component introduces additional connectors that are not defined in the constraining clause. Unless the connectors are part of a redeclared inherited top-level component, it is not possible that a user can connect to these newly introduced connectors. This is uncritical, if the not connected connectors do not have input variables, since the default connection semantic will set the flow variables to zero. Consequently, the restriction is introduced in Modelica 3.0 that additional connectors that are not defined in the constraining clause are default connectable, i.e. shall not have input variables.

A record or connector component that is directly replaceable or more commonly declared using a connector from a replaceable package also has a parameter dependent size. In such a case a redeclaration may add additional unknowns – which should also be balanced with matching equations. This is a "parameter-dependent size" and can be handled using the techniques in the next section – except that only unknowns are added in this way – but no equations, and at first it seems that this will inevitably lead to unbalanced models. However, it is possible to handle this correctly: by not using replaceable connectors directly, but instead use a replaceable package containing connectors and corresponding models (or functions) – similarly as in the Modelica.Media package.

This is a special case and we will not discuss the details. However, it has a direct relevance to a more fundamental change introduce in Modelica 3.0. Previously a connector component of a replaceable model component was implicitly replaceable, i.e. the problem that a redeclare could introduce missing equations was present for any replaceable model component having connectors – even if the connectors were not replaceable.

5 Parameter dependent sizes

An important aspect of the counting of equations is that it holds not only for the current set of parameters, but for any legal set of parameters values. The restriction in Modelica 3.0 is formulated such that even though the model should be balanced in all cases, the tool does not have to verify this. The reason was that *at the time* it was not possible to verify that a given set of restrictions ensures that submodels will always be within the restrictions, *and* that user libraries could be rewritten to conform to this.

Dymola can perform this test in several cases as will be outlined here; and in the remaining cases it is verified for the actual parameter values and a warning given.

The number of scalar variables is obtained by recursively symbolically adding the number of components of each variable:

- A scalar variable has the size 1.
- An array v[n] has the size: n·<the size of its elements>. Modelica implicitly assumes that $n \geq 0$. A multidimensional array is in Modelica considered as a nested array. For example, a matrix M[m, n] has the size m·n·<the size of its elements>. If the size is declared using the colon operator, v[:] = …, the size is represented as size(v, 1)·<the size of its elements>. The idea is to represent the size expressions of arrays symbolically as defined by the model developers.
- The size of a record is the sum of the size of its components.

The current restrictions when counting variables are evaluation of sizes of arrays of components and the conditions of conditional components.

The number of components of an equation is counted by traversing all its subexpressions and deducing the dimensionality and the size of each dimension and propagating this information upwards without any evaluation. At the top level the number of components is formed in a way analogous to that of variables. Also size constraints are collected for immediate or deferred checking. An interesting fact is that the size of a for-loop equation can be formed as the sum of the elements of an array constructor. One restriction is that the instantiation procedure may have evaluated some conditions.

The comparison of the number of variables and equations is done in 3 steps:

- First all variables which bindings cannot be modified (protect, final, constant) are substituted symbolically. If Dymola can symbolically deduce that the problem is balanced the check was successful in this respect. It means that the model is balanced irrespective of how a user rebinds or sets parameters that may be rebound or set.
- Otherwise, Dymola substitutes all non-literal bindings. If Dymola now can show that the problem is balanced, the comparison is finished. It means that the user can change parameter values that are literals, but not otherwise rebind parameter values without risking making the problem non-balanced. A remedy is to define critical parameter bindings to be final.

- The third step is to force evaluation of all size parameters and then compare. This is what Dymola has done previously when checking or translating a model.

As an example consider model Modelica.Blocks.-Continuous.StateSpace. The essence is:

```
block StateSpace
  parameter Real A[:, size(A,1)];
  parameter Real B[size(A,1), :];
  parameter Real C[:, size(A,1)];
  parameter Real D[size(C,1), size(B,2)]
        =zeros(size(C, 1), size(B, 2));
  extends Interfaces.MIMO(
                  final nin= size(B, 2),
                  final nout=size(C, 1));
  output Real x[size(A, 1)];
equation
  der(x) = A*x + B*u;
       y = C*x + D*u;
end StateSpace;
```

Checking the model in Dymola 7.0 results in

```
Model having the same number of
unknowns and equations:
  size(A, 1) + size(B, 2) + size(C, 1)
```

The counting of the unknown variables which are `x`, `u` and `y`, gives

```
size(A, 1) + nin + nout
```

The bindings for the parameters `nin` and `nout` are **`final`** and can be used for substitution, which gives the logged result. The counting of equations gives first `nin` for the inputs. The size check of

```
der(x) = A*x + B*u;
```

has to check possible size constraints for each subexpression. First, the matrix-vector multiplication, A*x requires `size(A,2)=size(x,1)`. Exploring the declarations of `A` and `x` shows that either side is equal to `size(A,1)`. The product `A*x` is a vector with `size(A*x,1)=size(A,1)`. The product `B*u` requires `size(B,2)=size(u,1)`. Exploring the declaration of `u` gives `size(u,1)=nin` and the final binding to nin gives `nin=size(B,2)`. Thus the constraint is fulfilled. The product `B*u` is a vector with `size(B*u,1)=size(B,1)`. Next, the sizes of the two terms `A*x` and `B*u` must be equal. They are both vectors and the size constraint is `size(A*x,1)= size(B*u,1)`. Since `size(A*x,1)=size(A,1)` and `size(B*u,1)=size(B,1)`and since the declaration states `size(B,1)=size(A,1)`, the constraint is fulfilled. The resulting sum is a vector of `size(A,1)`. Since the declaration of `x` specifies `size(x,1)= size(A,1)` all the size constraints of the equation are fulfilled symbolically for all allowed `A` and `B` matrices and it has `size(A,1)` components. Similarly the equation `y = C*x + D*u` is type consistent and has `size(C,1)` equations.

Dymola's facility for checking that two symbolic expressions are equal is rather elaborate. However, it cannot handle all cases such as complicated `for`-loop equations where there are, for example, conditions on the loop iterator. Dymola then resorts to numerical evaluation.

6 Limitations of the approach

The previous section shows that the rules in Modelica 3.0 make it possible to provide early checks of models that will avoid several hard to find errors when completing large models. The early checks are possible, since we only need the interface of sub-components. However, some errors are still possible when assembling sub-models and the natural question is why these errors cannot be handled in a better way.

6.1 Why are not all restrictions in the language?

As noted above we can prove that models are automatically balanced if built subject to certain restrictions, but not all of these restrictions are part of the language. This might seem odd considering that we want to ensure correctness early on, but it is necessary to allow textual (non-connect) equations to be given for low-level models. However, the check can still be performed at the same level – and uses the same description of balanced models; the only difference is that if the guidelines are followed the check becomes even simpler. When the guide-lines are not followed, a user would have to provide non-connector equations as a replacement for connections; thus for a model with only connector equations the simpler restrictions hold. This makes it straightforward to provide good diagnostics, while preserving the low-level openness of Modelica. Furthermore, a general recommendation is to avoid mixing connections and textual equations (see e.g. [5]); which makes it easier to separate the two cases.

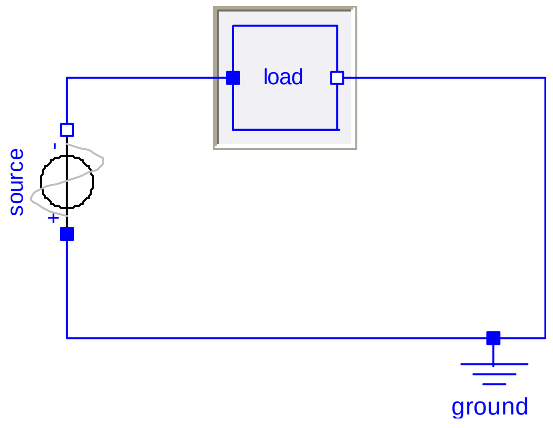
The examples where this is necessary include writing basic models such as a resistor where equations are given for the connectors - instead of adding additional connections, and correlations for media-models are built such that there are multiple potential inputs (see section 2.2 Correlations and non-connector inputs). Allowing equations for input connectors is also convenient in some other cases (e.g. when using table-lookup blocks); and by having the semantics above we avoid introducing a special semantic construct for defining the number of external equations needed for the media-models.

6.2 Requirements beyond balanced?

Having a balanced model is only a necessary requirement to be able to simulate it, but it is not sufficient. Whether a non-linear system of differential-algebraic equations has a solution is NP-hard in general; imposing restrictions to ensure a solution would impose such strict rules on the modeling (such as convex equations) that it would be not practical in general.

For the complete model a strong requirement is that the system of equations is structurally regular (i.e. no singularity when looking at the structure and ignoring the actual values). Dymola can also perform this check already for incomplete models and then use a generic coupling for local (replaceable) components, and for top-level connectors. This can provide useful diagnostics for many cases.

However, structural regularity is not entire well-defined, e.g. Dymola actually uses +/-1 from connections as well as zeros (finding more errors), and in contrast to balanced models structurally singular does not provide strong guarantees. Even plugging in parameters might turn a structurally regular system into a structurally singular one, since structurally singular only ensures that the equations are non-singular for “most” values of the non-zero elements. Obviously this also applies to redeclarations, especially since one often uses simplified models as test-cases. A simple example would be an electrical circuit testing different components with an ideal source:



This model (with an ideal voltage source) will become structurally singular if we plug in a short-circuit component as a load.

6.3 Restrictions on partial models?

Currently there are no balancing restrictions on partial models in Modelica, and the contents of base-classes are expanded prior to verifying the balancing. This is the formal semantics; a tool may internally handle this in a better way, taking special care of the non-trivial handling of connection sets, and of multiple inheritance of the same component.

The reason for the lack of restrictions is that the number of equations needed in derived models depend on whether the partial model is just an interface (e.g. TwoFlanges in the Rotational library – this just has two flange connectors), or contains an incomplete set of equations (e.g. Rigid in the Rotational library – which also specifies that the angles are identical). If we compare with e.g. Java this implies that a partial class may be either an interface or an abstract class.

A possible extension would be to have separate keyword for pure interfaces, and restricted such that only public connectors, parameters, and causal variables are present. The number of equations needed in derived classes would be uniquely defined from the interface. In that case it would not be necessary to verify that the interface is “balanced”, since it would follow automatically from the requirement on interfaces, and on connector classes.

A practical smaller extension would be to require that partial models may only be locally underbalanced, i.e. lacking equations, but not have too many equations.

These possible extensions have not yet been investigated in details.

7 Conclusions

The new restrictions in Modelica 3.0 make it possible to provide diagnostics earlier in the development process, while still maintaining the low-level openness of Modelica. These early diagnostics both shortens development time, and makes it possible to provide an interface for end-users where certain errors cannot occur – thus reducing the deployment and training cost for these users.

In Dymola 7.0, the restrictions introduced in Modelica 3.0 are supported, but are only imposed when using the Modelica Standard Library 3 (or later). This allows users to continue to run correct Modelica 2 models.

References

[1] **Modelica 3.0 Language Specification**, Modelica Association, September 2007. http://www.modelica.org/documents/ModelicaSpec30.pdf

[2] P. Bonus, P. Fritzson. **Automated Static Analysis of Equation-Based Components**. SIMULATION: Transactions of the Society for Modeling and Simulation Internal. Special issue on Component-based Modeling & Simulation. Vol 80:8, 2004.

[3] **Dymola**, by Dynasim AB, Sweden. See www.dynasim.se for more information.

[4] D. Broman, K. Nyström, P. Fritzson: **Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta**. In Proceedings of the 5th international conference on Generative programming and component engineering

[5] M. Tiller: **Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications**. In Proceedings of Modelica'2003 conference. http://www.modelica.org/events/Conference2003/papers/h31_parser_Tiller.pdf

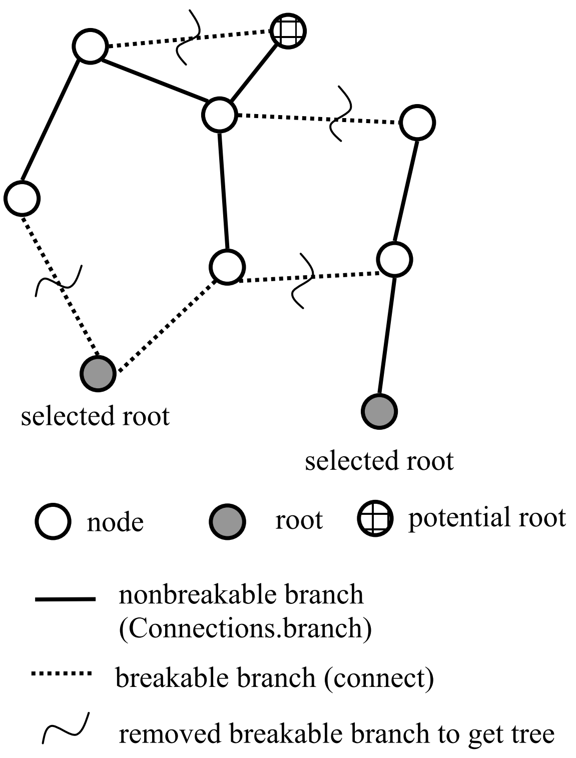
Appendix – Over-determined connectors

Over-determined connectors have been introduced in Modelica 2.1 to handle a certain class of consistently over-determined set of differential-algebraic equations, for example 3-dim. mechanical systems: Since a MultiBody connector contains the transformation matrix between the world frame and the connector frame, and there are constraints between the elements of a transformation matrix, connecting components with such a connector can lead to an over-determined (but consistent) set of unbalanced equations that have a mathematically well-defined solution. The over-determined connectors are defined and used in such a way, that a Modelica tool is able to remove the superfluous (consistent) equations arriving at a balanced set of equations, based on a graph analysis of the connection structure. For equation counting, it is of course important to take this special treatment into account:

A type class with an equalityConstraint(..) function declaration is called over-determined type. A record class with an equalityConstraint(..) function definition is called over-determined record. The equalityConstraint(R1,R2) functions are used to define the minimal number of equations stating that over-determined types or records R1 and R2 are identical. A connector that contains instances of over-determined type and/or record classes is called over-determined connector.

Every instance R_i of an over-determined type or record in an over-determined connector is a node in a virtual connection graph that is used to determine when the standard equation "R1 = R2" or when the equation "0 = equalityConstraint(R1,R2)" has to be used for the generation of connect(...) equations. The branches of the virtual connection graph are implicitly defined by "connect(..)" and explicitly by Connections.branch(...) statements. Additionally, corresponding nodes of the virtual connection graph have to be defined as roots or as potential roots with built-in functions Connections.root(...) and Connections.potentialRoot(...), respectively. Connections are treated as "breakable" branches. By removing appropriate breakable branches, the virtual connection graph is transformed into a set of spanning **trees**, each comprised of one root.

An example is given in the figure below, where all "dotted" lines characterize "`connect`(...)" equations. After building up the spanning trees, the connections that have to be removed to arrive at a spanning tree, are specially handled for the generation of the connection equations (see below):


selected root
selected root
node
root
potential root
nonbreakable branch (Connections.branch)
breakable branch (connect)
removed breakable branch to get tree

For potential roots the model tests if the root is selected, and then uses different equations. The flow-variables always give the same equations as normal connections, but for "potential" (=non-flow) variables this is different: If a `connect`(..) equation is not "broken", the standard equality equations hold. If a `connect(..)` equation is marked as "removed" in the virtual connection graph, less equations are provided by using the residual equations defined by the type or record specific function `equalityConstraint()` (shorted to `r()` below – with number of equations n_r) taking the two "potential" variables of the connected connectors as input arguments.

If we examine the same cases as in Figure 1 and consider `m1.c`, `m2.c`, `m3.c` as unconditional roots we get the same result for the left and for the right-case (the connection must be a removed branch since two unconditional roots are connected):

```
0 = m2.c.f + m3.c.f;    // nf equations
0 = r(m2.c.p, m3.c.p);  // nr equations
```

and thus we get:

left model (m1): $n_e^{root} = n_f$

right model (m2,m3): $2n_e^{root} = n_f + n_r$

or

$$n_e^{root} = n_f$$
$$n_r = n_f$$

If we instead have a potential root, we will in the left (unconnected) case select a root. If connected to a similar component one of them will get a root and the other one not, and the connection will not be broken (i.e. normal connection equations are introduced). We then get the size-constraint:

left model (m1): $n_e^{root} = n_f$

right model (m2,m3): $n_e^{root} + n_e^{non-root} = n_f + n_p$

Combing the two cases results in:

$$n_e^{root} = n_f$$
$$n_r = n_f$$
$$n_e^{non-root} = n_p$$

If there are several potential roots they should all give the same external equation count. The requirement on the residual size ($n_r = n_f$) is included in the Modelica specification [1], but additionally only the rooted external equation count is included ($n_e^{root} = n_f$) and not the non-rooted equation count.

Thus the balancing rules in Modelica 3.0 for over-determined connectors are incomplete and hold only for a very special case (e.g. when a MultiBody component is directly connected to the world object, which is a definite "root" of the virtual connection graph). This should be corrected in a future revision of the Modelica Specification. In the remaining part of this section, the non-rooted equation count is also taken into account.

In a similar way as in section 2.1 "Restrictions on physical connectors", the above derivation is also formulated in form of requirements on connectors and models (the derivation requires extending the above test models with a mixture of normal potential variables and potential variables of over-determined records or types):

1. The number of flow variables in a connector must be equal to the number of (normal) non-causal, non-flow variables + the number of residual equations of over-determined records and types (in the set of non-flow variables, the over-determined records and types are not included, because they are included via the residual equations)
2. The number of equations in a model = number of unknowns
   – number of inputs
   – number of flow variables
   – ((for every **Connections.Branch**(R1,R2)) and (for every **Connections.potentialRoot**(R1,..) where **Connections**.**isRoot**(R1) = **false**):
   number of R1 variables –
   number of R1 residual equations,
   i.e., the number of R1 constraint equations)
3. When using a model with over-determined connectors, i.e., making an instance, all missing equations of this component must be provided to make the component "balanced". Besides the standard rules, the only way to make over-determined connectors balanced is to connect to these connectors or by leaving them unconnected.

These rules shall be demonstrated at hand of the Modelica.Mechanics.MultiBody library:

The over-determined connector "`Orientation`" describes the transformation matrix from one frame to another frame:

```
record Orientation
 Real T[3, 3] "Transformation matrix";

 encapsulated function equalityConstraint
   import M=Modelica.Mechanics.MultiBody;
   input  M.Frames.Orientation R1
   input  M.Frames.Orientation R2
   output Real residue[3]
 algorithm
   residue := { ... }
 end equalityConstraint ;
end Orientation;
```

The `Orientation` object has a residue function with 3 equations and is used in a MultiBody connector to describe the rotation from the world frame to the connector frame:

```
connector Frame
  import SI = Modelica.SIunits;
  import M=Modelica.Mechanics.MultiBody;
  SI.Position   r_0[3] "Origin of frame"
  flow SI.Force f  [3] "Cut-forces"

  M.Frames.Orientation R  "Orientation"
  flow SI.Torque     t[3] "Cut-torque";
end Frame;
```

Connector frame has 3+3 = 6 flow variables (`f,t`), 3 normal, non-flow variables (`r_0`) and 3 residual equations (from `R`). Therefore, the connector fulfills rule 1 above.

FixedTranslational is a MultiBody model that translates one frame along a given position vector:

```
model FixedTranslation
  import SI=Modelica.SIunits;
  import M=Modelica.Mechanics.MultiBody;
  M.Interfaces.Frame_a frame_a
  M.Interfaces.Frame_b frame_b
  parameter SI.Position r[3]
    "Vector from frame_a to frame_b";
equation
  Connections.branch
               (frame_a.R, frame_b.R);
  frame_b.r_0 = frame_a.r_0 +
         M.Frames.resolve1(frame_a.R, r);
  frame_b.R = frame_a.R;
  zeros(3) = frame_a.f + frame_b.f;
  zeros(3) = frame_a.t + frame_b.t +
                     cross(r, frame_b.f);
end FixedTranslation;
```

The number of equations in `FixedTranslation` is required to be:

```
= 2*(3+3+9+3) // 2*(r_0+f+R.T+t)
  - 2*(3+3)   // 2*(f+t)
  - (9-3)     // (R.T - R.residuals)
= 18 equations
```

and the model fulfils this requirement.

`World` is the MultiBody model that defines the inertial frame as:

```
model World
  import M=Modelica.Mechanics.MultiBody;
  M.Interfaces.Frame_b frame_b;
equation
  Connections.root(frame_b.R);
  frame_b.r_0 = zeros(3);
  frame_b.R   = M.Frames.nullRotation();
end World
```

The number of equations in `World` is required to be:

```
= 3+3+9+3   // r_0+f+R.T+t
  - (3+3)   // (f+t)
= 12 equations
```

and the model fulfils this requirement.

`LineForce` is a MultiBody model that defines a force along a line between two frames. The difficulty is that if LineForce elements are directly coupled to each other, then the transformation matrix between two `LineForce` elements is arbitrary. This can be made mathematically well-defined, by setting one of the `LineForce` transformation matrices (= the selected root) to an arbitrary value:

```
model LineForce
  import SI=Modelica.SIunits;
  import M=Modelica.Mechanics.MultiBody;
  M.Interfaces.Frame_a frame_a
  M.Interfaces.Frame_b frame_b
    ...
equation
 Connections.potentialRoot(frame_a.R,10);
 Connections.potentialRoot(frame_b.R,10);

  frame_b.f = ...; // force law
  0 = frame_a.f + frame_b.f;

  if isRoot(frame_a.R) then
    frame_a.R = Frames.nullRotation();
  else
    frame_a.t = zeros(3);
  end if;

  if isRoot(frame_b.R) then
    frame_b.R = Frames.nullRotation();
  else
    frame_b.t = zeros(3);
  end if;
end LineForce;
```

The number of equations in `LineForce` depends on the selected roots. If **`isRoot`**(..) is **`false`** for both frames the number of equations are required to be:

```
= 2*(3+3+9+3) // 2*(r_0+f+R.T+t)
  - 2*(3+3)   // 2*(f+t)
  - 2*(9-3)   // 2*(R.T - R.residuals)
= 12 equations
```

and the model fulfils this requirement.

If **`isRoot`**(..) is **`false`** for one and **`true`** for the other frame, the number of equations are required to be:

```
= 2*(3+3+9+3) // 2*(r_0+f+R.T+t)
  - 2*(3+3)   // 2*(f+t)
  - 1*(9-3)   // 1*(R.T - R.residuals)
= 18 equations
```

and the model fulfils this requirement.