Enforcing model composability in Modelica

Sébastien Furic[1]
[1]LMS Imagine, France
sebastien.furic@lmsintl.com

Abstract

Modelica provides intuitive constructs to create and group *model definitions*. However, models themselves do not *compose*. In other words, the connection of type-compatible and locally balanced submodels does not generally yield a valid (e.g., balanced, structurally non-singular) model. Starting from simple examples of such invalid models (resulting from commonly encountered situations when using Modelica), this paper explains how those problems could be avoided by introducing a safer notion of physical connector, similar in some aspects to the VHDL-AMS notion of *terminal*. An extension of the notion of *connection* is also presented, providing new opportunities to make efficient use of ideal models in Modelica.

Keywords: model composition; high-level physical connector; effort variable; flow variable; connection graph; effort graph; flow graph

1 Introduction

A commonly encountered situation in Modelica when defining models by connection of submodels representing well-identified part of the whole design is that, even if each submodel has been checked for the absence of structural inconsistency, there is no guarantee that the result is itself structurally consistent. One may argue that this situation is normal since some combinations of models are "not physical". However, in the acausal modeling world, we know that one has to be careful about the "not physical" argument: for instance, "high-index" problems also are "not physical" from a certain point of view. But every experienced Modelica user knows that models yielding systems having non-minimal state can be given a meaning, and the result of accepting those models does not make Modelica an "everything runs anyway" simulation language: models still have well-defined semantics and Modelica can be used to express hard problems directly, without need for user assistance.

We claim that, given a proper notion of *model composition* (by generalizing the semantics of connections), we can, as in the case of models having non-minimal state mentioned above, give a sound meaning to a larger class of assemblies of type-compatible Modelica submodels. And fortunately the result is still not an "everything runs anyway" simulation language, but a more powerful one, where new kinds of models can be easily defined, offering Modelica new valuable possibilities.

2 Modelica modeling annoyances

Reading the documentation of the Modelica Standard Library reveals implicit assumptions made here and there, leading to difficulties when one has to use models in a given discipline. Let's consider for instance the Electrical library[1]: in the documentation we can read this definition of Modelica.Electrical.Analog.Basic.Ground:

"Ground of an electrical circuit. The potential at the ground node[2] is zero. Every electrical circuit has to contain at least one ground object."

The last sentence is rather confusing for the newcomer: why should every electrical model contain a "ground node"? After all, we all remember having in the past built paper-and-pencil electrical circuits without ground but for which it was possible to attribute unambiguously a meaning. Worse, the sentence seems to imply that sometimes *more than one* ground submodel has to be used: how many exactly for a given problem? And where to place them on the circuit? Let's make some experiments...

1 In the present paper, we will focus on the electrical domain, because models are often simple yet general enough to illustrate our thesis.

2 Notice the use of "node".

DOI: 10.3384/ecp09430023

2.1 A first example

Sometimes using Modelica results in frustrating experiences for the newcomer: even a simple R circuit may not simulate! Figure 1 below shows the diagram of such a circuit built under Scicos[3]:

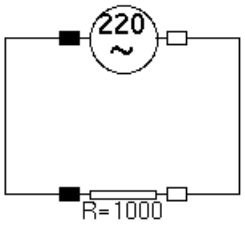

Figure 1: A naive R circuit

The system seems to be well defined: a voltage source imposes a voltage difference between the pins of a resistor and the resistor imposes the current flowing into the circuit. We can verify that by inspecting the equivalent "flat" model that, according to Modelica semantics, is:

```
model RCircuit
  Real src.p.v;
  Real src.p.i;
  Real src.n.v;
  Real src.n.i;
  Real src.v;
  Real src.i;
  Real res.p.v;
  Real res.p.i;
  Real res.n.v;
  Real res.n.i;
  Real res.v;
  Real res.i;
equation // generated by Source
  src.v = src.p.v - src.n.v;
  src.p.i + src.n.i = 0;
  src.i = src.p.i;
  src.v = 220 * sin(314.15 * time);
equation // generated by Resistor
  res.v = res.p.v - res.n.v;
  res.p.i + res.n.i = 0;
  res.i = res.p.i;
  res.v = 1000 * res.i;
equation // connection equations
  src.n.v = res.n.v;
  res.p.v = src.p.v;
  src.n.i + res.n.i = 0;
  res.p.i + src.p.i = 0;
end Rcircuit;
```

That system captures the required constraints. Indeed from:

```
src.v = src.p.v - src.n.v;
res.v = res.p.v - res.n.v;
src.n.v = res.n.v;
res.p.v = src.p.v;
```

we deduce, as expected, that:

```
src.v = res.v;
```

And then, from:

```
res.v = 1000 * res.i;
src.v = 220 * sin(314.15 * time);
```

we found that:

```
res.i = 0.220 * sin(314.15 * time);
```

Then by exploiting:

```
res.i = res.p.i;
res.p.i + src.p.i = 0;
src.i = src.p.i;
```

we deduce that:

```
src.i = -res.i;
```

Everything seems fine... However the simulation of that simple model fails miserably. Why? The careful reader may have noticed that, in order to resolve the above system for the "variables of interest" (`src.v`, `res.v`, `src.i` and `res.i`), we did not use all the constraints. If we would have done so, we would have found that the unused constraints:

```
src.n.i + res.n.i = 0;
src.p.i + src.n.i = 0;
res.p.i + res.n.i = 0;
```

coupled with the fact that:

```
res.i = res.p.i;
src.i = src.p.i;
```

would have lead to an over-constrained (but consistent) system of equations having five equations and only four unknowns. Also, we did not solve the system for all the variables: `src.p.v`, `src.n.v`, `res.p.v` and `res.n.v` remain undetermined. Trying to determine their value leads to the opposite problem we encountered while solving for the flow variables: the subsystem is under-constrained.

From the mathematical analysis point of view, our system has a *singular jacobian matrix*. However, that singularity is not the result of unfortunate values taken by the coefficients of the matrix: even the *incidence matrix*[4] of the system is singular.

3 Freely available modeling tool with Modelica capabilities (http://www.scicos.org).

4 A matrix having the same size as the jacobian matrix, and whose coefficients indicate eventual contributions of each variable to the corresponding jacobian matrix

Like many other newcomers the user that built the model on Figure 1 would probably be said that she/he has overlooked the advice given in the documentation of Modelica.Electrical.Analog.Basic.Ground: indeed, adding a ground submodel to the circuit would have saved the situation by introducing the missing voltage equations and one degree of freedom for the current needed for the calculation of all the absolute voltages (that we don't need to know) and "outgoing currents" (that we don't need to know too).

Things learned from that error are:

- using Modelica libraries implies learning (sometimes implicit) rules *that are not enforced at the language level*
- Modelica forces users to give equations to compute unneeded quantities

2.2 Ideal models

Another look at the documentation that comes with the Electrical library reveals that the ground submodel is not the only one that carries out structural modeling assumptions. For instance, the description of Modelica.Electrical.Analog.Ideal.IdealOpeningSwitch includes the following warning:

"In order to prevent singularities during switching, the opened switch has a (very low) conductance Goff and the closed switch has a (very low) resistance Ron. The limiting case is also allowed, i.e., the resistance Ron of the closed switch could be exactly zero and the conductance Goff of the open switch could be also exactly zero. Note, there are circuits, where a description with zero Ron or zero Goff is not possible."

It is legitimate to ask oneself what is a *low conductance* (resp. *low resistance*). Also, which are those circuits that disallow zero Ron and/or zero Roff?

The difficulty when one has to determine the appropriate conductance (resp. resistance) of a nearly-ideal opening switch is that the answer depends both on the circuit itself and on the compiler/solver pair.

Consider a model having several switches in parallel (for instance, a model of a fault-tolerant circuit in a nuclear plant): it is well-known that the two-pin model equivalent to those switches put together is a variable resistor whose resistance is given by the product of the variable resistances of the individual switches divided by the sum of those same resistances. It follows that small (resp. big) values of the individual resistances may lead to tiny (resp. huge) values for the resistance of the equivalent two-pin model. This implies that for large electrical models, where one cannot predict the sequences of openings/closings of switches, it is nearly impossible to determine with a high degree of confidence a suitable value for the resistances of the switches, even if one got the complete source code describing the circuit model.

Concerning circuits where a description with zero Ron or zero Goff is not possible, experimenting a little with the language rapidly convinces us that the *connection* of the graph of linked model instances is a key property, and it brings us back to the problem of effort reference depicted in section 2.1: each *connected component* of a circuit has to contain one `Ground` instance. Consider the following circuit:

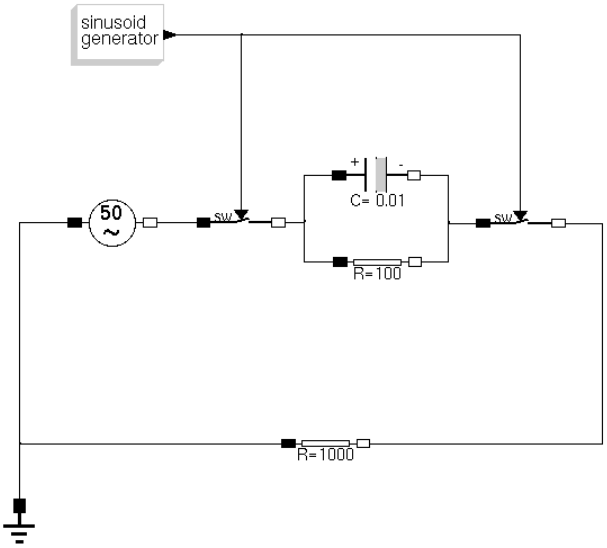

Figure 2: A circuit with switches

In this circuit, the two coupled opening switches control the connection of two subcircuits: when the switches are closed the capacitor is charged, and when the switches are opened, the capacitor discharges itself through the resistor connected in parallel with it.

What happens if we use ideal switches? If both switches are opened, we get two subcircuits, one of them having no `Ground` instance attached. We are then in the same situation as depicted in Figure 1: the solver fails due to the structurally singular jacobian matrix. But if we add a `Ground` instance as in Figure 3 below, the simulation can be run... until both switches are closed, in which case we have a connected graph of model instances with two effort ref-

coefficient: zero indicates no contribution and one indicates a (possibly null) contribution.

erences. Removing the added `Ground` instance and forcing the switches to remain closed allows the model to be simulated. So we are in a situation where, in order to be able to simulate our model, we have either to put or to remove a `Ground` instance from the circuit, depending on the state of the switches.

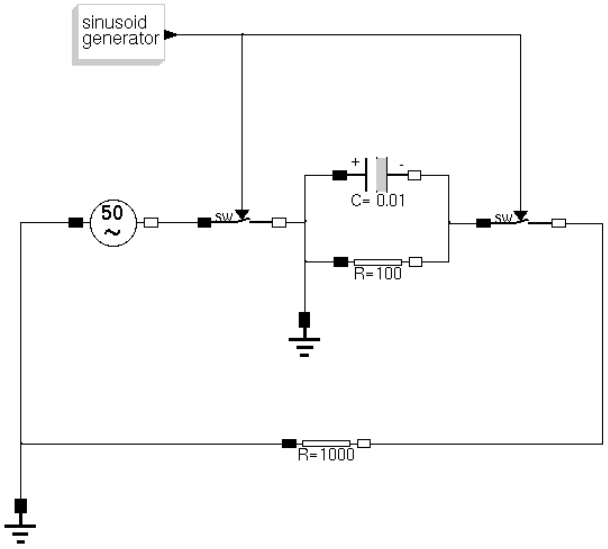

Figure 3: A modified version of the circuit with switches

The conclusion from experiments with ideal models could be that one should avoid using them because they lead to under- and over-constrained subsystems of equations. This is not the correct conclusion in our opinion: we think that ideal models are extremely useful in some situations (e.g., where one needs to save computation effort for instance) and that the modeling language should be able to properly handle them by yielding simulation code that conforms to our expectations.

2.3 Connection variables

Note: Without loss of generality, we deliberately consider here the simple pattern of a main model built out of submodels connected together. Submodels themselves are supposed to contain only simple equations (i.e., equalities). Indeed, the purpose of this paper is not to discuss multiple ways to build Modelica models but to illustrate problems with the current approach, based on use of first-class connection variables.

Information exchange between models in Modelica is usually performed by means of *connectors* and *connections*. Connectors are aggregations of *connection variables*, divided into *flow* and *potential* variables. The connection of several compatible connectors expresses constraints between connection variables that match:

- values of potential variables have to be equal
- values of flow variables have to sum to zero

A special case exists for unconnected connectors:

- values of potential variables are not constrained
- values of flow variables have to be zero

From the outside of submodels point of view, Modelica provides a rather high-level construct (connection equations) to express connections of models without having to manipulate connection variables directly. That approach offers several advantages:

- the code reflects the topology of the model, so the code is easy to understand
- it is possible to add a new branch to a model without having to worry about the consequences on the new connection constraints, so the code is easy to extend
- it is impossible to break fundamental invariants attached to both kinds of connection variables, so the code is robust

In contrast, from the inside of submodels point of view, things are not as simple and clear: users have to manipulate connection variables directly, and, as a consequence, nothing prevents them to violate fundamental invariants such as flow preservation. Worse: the compiler has no way to prove that models preserve those invariants since, inside submodels, everything is ultimately an ordinary equation.

That situation is a bit odd: after all, why not having the counterpart of connection equations inside submodels? Having a construct that would handle connection variables automatically the way **`connect`** does outside submodels not only would help users to write correct (and machine-checkable) models, but also would lead to more general models. Indeed, the fundamental problem with the connection variable approach is that it depends on the assumption that connection variables always exist in the sense that there is always enough constraints in the model to define all of them. That assumption is false, as demonstrated in the previous sections, and the goal of ground-like models (to be provided by users!) is precisely to compensate the lack of potential constraints and the excess of flow constraints that arise naturally in connection variable-based semantics.

At this point, it is legitimate to ask ourselves why not simply make connection variables second-class citizens of the language and let higher-level constructs

directly deal with them. Indeed, this would be a solution: if for instance the compiler would introduce connection variables on demand (i.e., just as many connection variables as necessary to solve the problem for variables of interest) we would avoid the problems encountered so far. But let's think a bit more about those connection variables: what kind of information do they carry under the assumption that you can't directly access them? The answer is rather simple: nothing meaningful for the user. In consequence, the thesis we develop in this paper goes even further: we simply advocate a language with no connection variables at all in the physical domain.

3 Proposal for an enhanced Modelica approach

3.1 The example of VHDL-AMS

Several modeling languages already adopted a theoretical model with no connection variables[5]. Among them, VHDL-AMS[2] is probably the most famous one in the Modelica community. In this section, we present an quick overview of the way VHDL-AMS handles connection of analog submodels.

VHDL-AMS defines *terminals* to represent physical connection points (the equivalent of Modelica connectors, in the analog domain). A terminal is declared to be of a given *nature*, i.e., energy domain. Here is an example of declaration of an electrical nature:

```
subtype voltage is real;
subtype current is real;
nature electrical is
  voltage across
  current through
  ground reference;
```

The first two lines are just for convenience: they define two subtypes of `real` (the equivalent of Modelica's `Real` type) used to represent voltages and currents, respectively. More interesting is the declaration of `electrical`:

- `voltage across` declares the *across type* associated with `electrical`
- `current through` declares the *through type* associated with `electrical`
- `ground reference` names the *reference terminal*, a special terminal that holds the reference potential associated with `electrical`

We can observe that no name is introduced for neither the across type nor the through type. Only `ground`, the name of the reference terminal, is introduced by the definition of `electrical`: in VHDL-AMS there is no need to introduce any ground submodel since the ground is a connection point, not a submodel[6].

Natures can be used to define terminals using the following syntax:

```
terminal p, n: electrical;
```

The above line declares two terminals `p` and `n` having `electrical` nature. How do we access across and through values held by the terminals in VHDL-AMS? Here is the trick: we cannot. As written above, terminals only represent physical connection points and, as such, they do not stand for placeholders for any kind of connection variables. The only way offered by VHDL-AMS to access across and through quantities inside the equivalent of submodels is by means of *branch quantity* declarations:

```
quantity
  v across
  i through
  p to n;
```

The above declaration reads as follow:

- `v across` declares a variable having the across type of `p` and `n`, that holds the value of the voltage difference between `p` and `n`
- `i through` declares a variable having the through type of `p` and `n`, that holds the value of the current that flows from `p` to `n`

We can notice in the declaration above that nothing requires the existence of any kind of connection variable, as expected. Indeed, the declaration really just states that:

- the *across quantity between* `p` *and* `n` can be refered to as "`v`"
- the *through quantity flowing from* `p` *to* `n` can be refered to as "`i`"

Branch quantity declarations coupled with *port maps* (the equivalent of Modelica's connection equations in VHDL-AMS) define a connection graph that is used to elaborate the missing constraints in VHDL-AMS models. Which advantages do they provide over the Modelica approach? First, inside submodels,

5 Simscape, from The MathWorks™, offers an intermediate level of abstraction (connection variables are visible as port attributes) but users are strongly encouraged not to use them directly.

6 Remember footnote 2.

we get the conciseness and the clarity already offered outside submodels in Modelica by means of connection equations: the code is easy to read, easy to maintain and more robust (compare the branch quantity declaration given above in VHDL-AMS with the definition of partial "two-pin" submodels in Modelica libraries). Second (and the most important regarding the subject of this paper), thanks to those high-level constructs, **a VHDL-AMS compiler has a global view of the whole connection graph**:

- concerning potentials: where a Modelica compiler only knows locally that absolute potentials in a connection set should be equal, a VHDL-AMS compiler also knows which loop(s) of the connection graph a given potential contributes to (this allows circuits as in Figure 1 to be successfully compiled)
- concerning flows: where a Modelica compiler only knows what happens locally in a connection set, a VHDL-AMS compiler also knows how flows traverse submodels (since, inside submodels, flows are explicitly declared as such) and, consequently, how flows traverse the whole connection graph.

The strength of VHDL-AMS regarding analog modeling comes precisely from that global view of the connection graph: **a VHDL-AMS compiler can fully apply both *Kirchhoff laws*[7] to the whole system** whereas a Modelica compiler can only apply the first one, and moreover, only partially (actually, to each connection set). It follows that a VHDL-AMS compiler can provide the exact number of missing constraints to solve the whole system without introducing any connection variable and without resorting to the "manual adjustments" (e.g., positioning ground-like submodels) required by Modelica.

Undeniably, VHDL-AMS outperforms Modelica here. Alas, there are still some limitations: models such as depicted in Figure 3 still remain impossible to define... The reason is that VHDL-AMS requirements for solvability (see [2]) impose one equation to be present in ideal switches, so users have to state explicitly that the current flowing through the submodel is zero when in "open" mode[8]. Global flow analysis finds that our VHDL-AMS model with two switches is well-structured whereas two constraints may dynamically impose the value of the flow on the same branch of the circuit. We are in the same situation encountered in Modelica while solving the simple R circuit: under some switching conditions, the incidence matrix of the system is singular.

We conclude from those observations that having high-level constructs such as VHDL-AMS branch quantity declarations in an acausal modeling language greatly enhances expressiveness. However, that approach, even if better than what Modelica currently provides, does not suffice to resolve all the issues. In the next sections, starting from a variant of branch quantity declarations seen above, we propose an enhancement over VHDL-AMS, that solves the issues encountered in models involving ideal submodels.

3.2 High-level physical connectors

In order to benefit from the full power of global connection graph analysis in Modelica, we need to be able to define the equivalent of VHDL-AMS terminals and branch quantities. In the case of electrical modeling, it would give something like:

```
type Voltage = Real(unit="V");
type Current = Real(unit="A");
connector Pin
  across Voltage;
  through Current;
end Pin;
```

Several comments need to be made at this point. First, unlike in VHDL-AMS, we do not begin by defining a nature to be subsequently used in terminal or branch quantity declaration. Instead, we define directly a *high-level physical connector* for the physical domain of interest. This avoids introducing too many new language constructs (and keywords) while still remaining general enough, as we will see below. Second, we impose that any high-level physical connector contains exactly one across type definition and its associated through type definition. It is however possible to add traditional connection variables, parameters, etc., in the same connector definition, in the pure Modelica spirit. Also, high-level physical connectors can be aggregated into enclosing connec-

7 The first law, also called *Kirchhoff's junction rule*, states that the directed sum of the flows at every node of a circuit is zero. The second law, also called *Kirchhoff's loop (or mesh) rule*, states that the directed sum of the efforts around any closed circuit is zero.

8 VHDL-AMS features *generate statements* to generate constraints conditionally, so one might hope to use them here... Alas, the switching condition is dynamic in our case and generate statements are limited to static cases. Also, we don't know, at submodel creation time, under which condition constraints have to be present (or not).

tors, like any other kind of Modelica connector. Third, the careful reader may have noticed that we did not define any *reference connector* when defining `Pin` above. The main reason is that we don't want to pollute name spaces with new names, especially when context always permits to disambiguate the code. Indeed, references to the reference connector always occur in a connection statement. For obvious reasons, we impose that at least one of the connected entities has to be an instance of a compatible connector type[9] other than the reference connector. In consequence, we propose that the keyword **`reference`**, used in any connection statement in place of a connector reference, represents the *ad-hoc* type-compatible reference connector.

3.3 Connections and effort/flow variables

In order to conveniently express connections in a natural Modelica style, we propose the usage of the keyword **`connect`** to be extended in two ways:

- applied to high-level physical connectors[10], it is used to express that its arguments have to be considered as the same physical connection point, or *node*
- applied to more than two connector references, it enables users to enhance readability of connection statements by allowing local connection sets to be expressed directly[11]

We also propose to declare the equivalent of branch quantities by means of two separate constructs:

```
across(v, p, n);
through(i, p, n);
```

The first line reads "`v` denotes the potential difference between `p` and `n`": it is used to declare an *effort variable* (the first argument) holding the effort difference measured between the "plus" and "minus" connectors denoted respectively by its second and third arguments. Notice that we do not declare the type of `v`: instead we impose that at least one of the two connector arguments to **`across`** has to reference a user-defined instance of a high-level physical connector, so the type can be deduced from context (it is of course the across type of the corresponding high-level physical connector type). Similarly to **`across`**, **`through`** introduces a new variable, but in that case the introduced *flow variable* holds the flow quantity flowing from the "plus" connector to the "minus" connector[12].

3.4 Sources and sensors

Our theoretical model is not yet expressive enough: pure sources and pure sensors are missing. The *raison d'être* of those entities is to enforce conceptual decoupling of the effort/flow world from the signal world by enabling users to express input/output constraints between both worlds. This eases documentation, model analysis and debugging.

Sources and sensors are defined by means of *tees*. The proposed syntax for sources, or *input tees*, is:

```
across(input v_in, p, n);
through(input i_in, p, n);
```

The two lines above introduce respectively and effort source and a flow source. The corresponding effort/flow is constrained from the outside world by means of the type-compatible input connector signal whose name follows **`input`**. Similarly, the syntax for sensors, or *output tees*, is:

```
across(output v_out, p, n);
through(output i_out, p, n);
```

Unsurprisingly, the two lines above introduce respectively an effort sensor and a flow sensor. The type-compatible output connector signal is constrained by the corresponding effort/flow.

The theoretical model introduced here differs from VHDL-AMS's one: in particular, we allow the declaration of pure effort sources, whereas VHDL-AMS forbids them[13]. At this point, we have just defined a variant (with minor enhancements) of what VHDL-AMS already proposes, but we still have to solve the issues encountered with the use of ideal submodels.

3.5 Conditional connections

We have seen in previous sections that submodels such as ideal switches introduce dynamic *structural singularities (i.e., the incidence matrix of the system*

9 Remember that Modelica features a *structural* type system.

10 Of course having compatible types, which actually means "same types" here due to type invariance imposed by acausal semantics.

11 This should not preclude connection sets to be split over several connection statements, however.

12 Notice that the decoupling of across and through variable declarations enables decoupled sign conventions.

13 That limitation of VHDL-AMS probably has its roots in the way *structural sets of equations* (i.e., VHDL-AMS's equivalent of connection equations) are required to be elaborated: by means of an application of the *Modified Nodal Analysis* (see [3]).

becomes singular under some circumstances that happen at unpredictable times during simulation). This suggests that a possible way to circumvent those problems could be to dynamically adjust the constraints to be solved instead of trying to hide numerical problems into equations that, as a result, introduce stiffnesses and numerical singularities into systems of equations. We then propose here a generalization of connection equations called conditional connections *and explain why this kind of construct can be used to solve issues encountered so far.*

Conditional connections are just *guarded* connect statements in Modelica programs: syntactically, we simply allow connect statements to appear in a branch of a conditional statement[14]. The semantics are naturally generalized to allow dynamic changes in the connection graph of a model: whenever a conditional construct activates a connect statement during simulation, the connection constraints get updated in accordance. Here is the code of an ideal switch using conditional connections:

```
model Switch
  Pin p, n;
  input Boolean on;
equation
  if on then
    connect(p, n);
  end if;
end Switch;
```

It is interesting to notice that the "if" clause above is not "balanced". Indeed, its only purpose is to specify conditions under which the topology of the physical connection graph of models containing instances of `Switch` changes[15].

It is straightforward to see why conditional connections solve issues encountered with the use of ideal switches. Indeed, models such as `Switch` above do introduce equations only on demand[16], to satisfy both Kirchhoff's laws: we have just defined a dynamic version of an elaboration method *à la* VHDL-AMS. It follows that, for instance, having several closed switches in parallel or several open switches on the same branch of a circuit is no longer a problem, provided we know how to cope with more general connection constraints. An elaboration algorithm that satisfies those requirements is presented in the next section.

4 Generation of connection equations

4.1 Physical connection graphs, across graphs and flow graphs

Across declarations and through declarations not only introduce a new identifiers, they also introduce new edges in a structure called a *physical connection graph* of the model. That directed graph is built as follow:

- vertices represent *connection sets* of high-level physical connectors[17] of the original model
- edges represent either across declarations or through declarations of the original model. Direction information is preserved: edges are directed from the connection set which the positive high-level physical connector belongs to to the connection set which the negative high-level physical belongs to. Also, causality is preserved, i.e., input/output information associated with tees is represented in the graph

It is important to notice that physical connection graphs are not explicit in the original model structure. However, they can be seen as a kind of dual of the original model structure considered as a graph, for a given configuration of conditional connections: vertices in the original model (i.e., submodels containing only simple equations) are used to build edges in the physical connection graph, and, conversely, edges in the original model (introduced by means of **`connect`***) are used to build vertices in the physical connection graph. Notice also that we associate exactly one physical connection graph with a model for each configuration of the conditional connections. This implies that physical connection graphs are generally disconnected: indeed, the associated model may eventually contain transformers and gyrators, and make use of several physical domains, for example.*

A physical connection graph can be seen as the superimposition of two simpler graphs having the same

14 *When* and *if* clauses should be equally considered here. But due to the lack of a rigorous hybrid theoretical model in Modelica currently, we will focus on continuous-time equations and if clauses only in the focus of this paper.

15 After all, since our aim is to equip Modelica with composable submodels, it is not so surprising that composition statements themselves do not require balancing constraints!

16 In particular, a model composed of ideal switches only does not contains any variable nor equation.

17 The reference connector is view as an ordinary connector here.

vertices: the *effort graph* and the *flow graph*. Edges of the effort (resp., flow) graph represent across (resp., through) declarations of the original model. Figure 4 below shows a representation of the connection graph corresponding to the model in Figure 1.

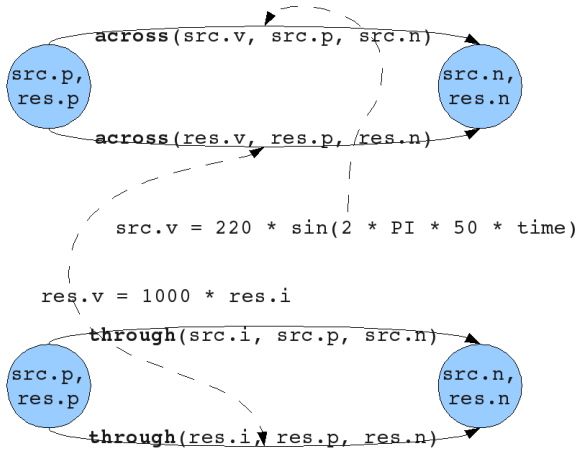

Figure 4: a simple connection graph

The effort graph is represented at the top of the figure and the flow graph at the bottom. Blue vertices represent connection sets (shared between both the effort graph and the flow graph). Directed edges between two blue vertices represent across or flow definitions, depending on the nature of the subgraph they belong to. For information, relations between coupled effort and flow variables are represented by dashed arrow-ended curves labelled with the corresponding explicit equations.

4.2 Elaboration algorithm

The elaboration algorithm we propose in this paper operates by determining effort constraints and flow constraints independently. Notice that since we want to enable dynamic changes in the topology of the connection graph associated with a model, the algorithm should be applied to each configuration required by the simulation[18].

Effort constraints are determined this way:

- for each connected component of the effort graph, perform a depth-first traversal with marking
  - for each detected loop, generate the sum-to-zero of effort variables along the loop by following this sign convention: effort variables associated with edges oriented in the direction of the loop traversal are counted as positive and the other ones as negative.

Flow constraints are determined this way:

- for each connected component of the flow graph, perform a depth-first traversal with marking
  - for each detected loop with at least two vertices[19], generate, if not already done[20], the sum-to-zero of flow variables at each vertex along the loop until all the variables of the loop have been used at least once. The following sign convention is used for summation: flow variables associated with incoming edges are counted as positive and the other ones as negative
- generate equations constraining flow variables associated with edges that do not belong to any loop to have a null value.

The proof of the algorithm is omited but we give here the general idea behind it:

- a loop of length *n* where each pair of high-level physical connectors is connected exactly by one across definition and one through definition in the physical connection graph yields *1* effort constraint (the directed sum of effort variables equals zero) and *n - 1* flow constraints (the directed sum of flow variables is generated at each vertex but one[21])
- it follows that the final system of equations has *2n* unknowns (*n* effort variables and *n* flow variables) and *2n* equations (*n* equations explicitly given by the user, as required by balancing constraints, and *n* equations automatically introduced by the elaboration algorithm).

5 Example

In order to illustrate the power of high-level physical connectors, let's try to define the model depicted in Figure 2. We will reuse the definitions of `Pin` and

18 Typically, each time the branch of a conditional equation becomes active. But a simulation environment may optimize simulation time by anticipating configurations, by caching old configurations, etc. Those optimization strategies are beyond the scope of this paper.

19 Loops with only one node should not yield flow constraints.

20 Since the same vertex may belong to several loops.

21 Flow variables associated with the ignored vertex have been already used in equations associated with its neighbors, so no equation is generated for that vertex.

`Switch` previously introduced in 3.2 and 3.5, respectively. The model also requires the definition of a voltage source, a resistor and a capacitor. They are given here:

```
model VoltageSource
  constant Real PI = acos(-1);
  parameter Real V0;
  Pin p, n;
  across(v, p, n);
  through(i, p, n);
equation
  v = V0 * sin(2 * PI * 50 * time);
end VoltageSource;

model Resistor
  parameter Real R;
  Pin p, n;
  across(v, p, n);
  through(i, p, n);
equation
  v = R * i;
end Resistor;

model Capacitor
  parameter Real C;
  Pin p, n;
  across(v, p, n);
  through(i, p, n);
equation
  C * der(v) = i;
end Capacitor;
```

Notice the conciseness of those definitions, thanks to the use of high-level physical connectors: no inheritance from an abstract `TwoPin` class is needed. Also, the respective roles of `v` and `i` are explicit in the code: this greatly helps understanding the physics behind submodels. The `VoltageSource` submodel deserves a special comment: despite the absence of `i` in the equation of the submodel, we have to define it because otherwise it would make the submodel a pure voltage source and, as a consequence, the current would not traverse it. Since we want a two-pin-like submodel, we have to define the effort/flow pair of variables.

Just for fun, we can also define a (useless) ground submodel:

```
model Ground
  Pin p;
equation
  connect(p, reference);
end Ground;
```

Notice that, contrary to Modelica's traditional ground submodel, our ground submodel does not introduce any explicit equation (nor any variable at all) in models into which it is used.

The circuit (with logic of switches omited) can then be defined as:

```
model Circuit
  Ground gnd;
  VoltageSource src(V0=50);
  Resistor res1(R=1000);
  Resistor res2(R=100);
  Capacitor cap(C=0.01);
  Switch sw1(on=...), sw2(on=...);
  ...
equation
  connect(gnd.p, src.p, res1.p);
  connect(src.n, sw1.p);
  connect(sw1.n, cap.p, res2.p);
  connect(cap.n, res2.n, sw2.p);
  connect(sw2.n, res1.n);
  ...
end Circuit;
```

We will study the two possible configurations where, respectively, both switches are open and both switches are closed.

The first configuration yields the physical connection graph depicted in Figure 5 below. Both effort and flow graph have the same topology because we only used two-pin-like submodels to hold user-defined equations.

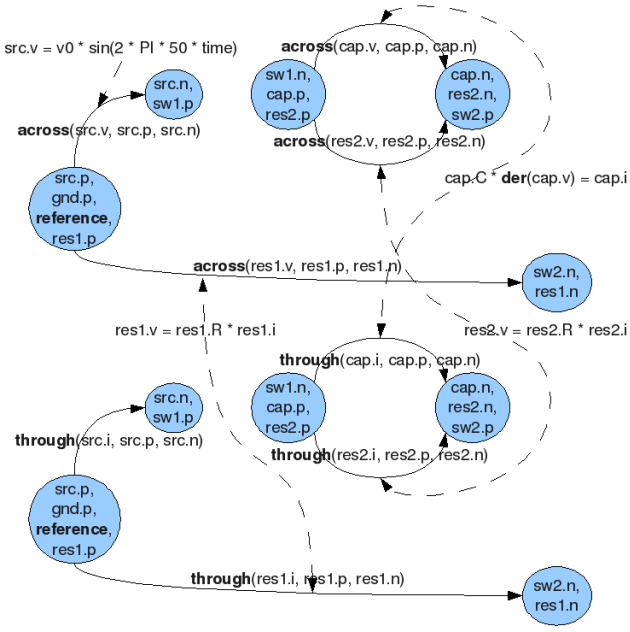

Figure 5: physical connection graph corresponding to model in Figure 2 in "open" mode

Applying the elaboration algorithm to the physical connection graph gives for instance (depending on the order into which vertices are examined):

```
cap.v - res2.v = 0;
src.i = 0;
res1.i = 0;
cap.i + res2.i = 0;
```

Only one effort constraint has been created, since the effort graph only has one loop. Three flow constraints have been created: two for the the leftmost subgraph (which contains no loop) and one for the rightmost subgraph (which is a loop with two vertices). We finally get a system with eight unknowns and eight equations once the four explicit equations are merged with the automatically generated ones.

The second configuration yields the physical connection graph depicted in Figure 6 below.

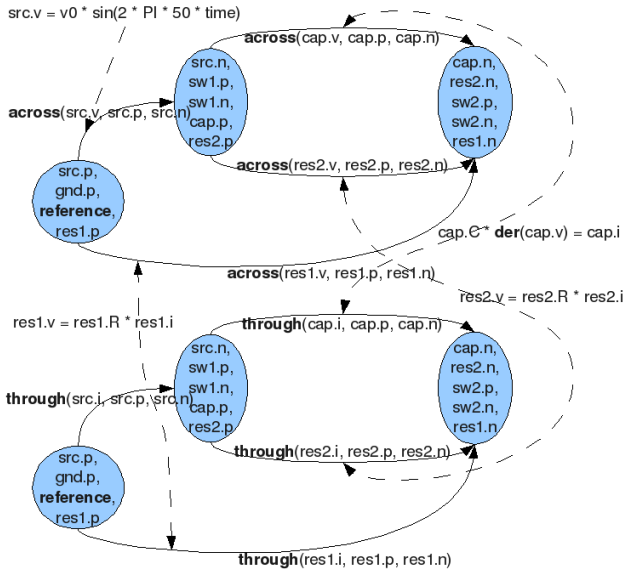

Figure 6: physical connection graph corresponding to model in Figure 2 in "closed" mode

Applied to that graph, our elaboration algorithm gives for instance:

```
res1.v - res2.v - src.v = 0;
res2.v - cap.v = 0;
-src.i - res1.i = 0;
cap.i + res2.i + res1.i = 0;
```

Again, a system with eight unknowns and eight equations is finally produced. Two effort constraints and two flow constraints have been generated, one for each loop in both the effort graph and the flow graph.

It is interesting to notice that in both cases only eight equations are produced where the equivalent traditional Modelica program would have produced 38 equations! (each two-pin-like submodel introduces six variables and the ground submodel, two). Also, equations generated by our algorithm always correspond to physical properties of the model: instead of obfuscating the final system of equations like traditional connection equations do, equations generated by our algorithm may be helpful to users to analyse their models, and, eventually, to debug them.

6 Conclusion

In this paper, we have proposed a solution to enforce composability of Modelica submodels: any composition of (possibly ideal) submodels that is physically sound now yields a non-singular system of equations. Now it would be interesting to exploit the additional syntactic information offered by our proposal to better statically typecheck models: it should help detecting errors more accurately than a solution based on traditional Modelica connectors, especially if coupled with a method that would take incidence information into account[22]. Also, and more importantly, we hope that, coupled with a rigorous hybrid theoretical model, our proposal would serve as a basis to get rid of several special-purpose or composition-unfriendly features of Modelica designed to cope with limitations in expressiveness, especially the recent Stream proposal but also the Overconstrained Connection-Based Equation Systems proposal, among others[23]. Another interesting application would be a type system that would be powerful enough to ensure true "plug compatibility": separate compilation of submodels would become possible and intellectual property protection would directly benefit from that.

22 Notice that explicit incidence information does not violate intellectual property!

23 As the first sentence of the Revised[5] Report on the Algorithmic Language Scheme nicely says: *"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."*

7 Acknowledgments

This work is part of the contribution of LMS to the ITEA2 European project OpenProd[24]. Many thanks to my colleagues from LMS Imagine for their help and suggestions, especially El Djillali Talbi, Antoine Viel, Loig Allain, Yohan Broussaud, Olivier Broca and Denis Fargeton. Also, many thanks to Michael Tiller from Emmeskay, Wilfrid Marquis-Favre from INSA-Lyon and David Broman from Linköping University: discussions with them are always extremely valuable.

References

[1] The Modelica Association, ***Modelica Standard Library 3.0***, PDF Documentation, available at http://www.modelica.org/libraries/Modelica/releases/3.0.1/ModelicaStandardLibrary_3_0_Documentation.zip

[2] P. J. Ashenden, G. D. Peterson, D. A. Teegarden, ***The System Designer's Guide to VHDL-AMS — Analog, Mixed-Signal, and Mixed-Technology Modeling***, Systems On Silicon, Morgan Kaufmann, 2003, ISBN 1-55860-749-8

[3] C. Tischendorf, ***Topological Index Calculation of DAEs in Circuit Simulation***, 1997

[4] G. Dauphin-Tanguy, ***Les bond graphs***, HERMES Science Europe Ltd, 2000, ISBN 2-7462-0158-5

[5] P.J.L. Cuijpers, J.F. Broenink, P.J. Mosterman, ***Constitutive Hybrid Processes: a Process-Algebraic Semantics for Hybrid Bond Graphs***, in SIMULATION, Vol. 84, Issue 7, pp 339-358, 2008

24 See http://www.ida.liu.se/~pelab/OpenProd/

