Node-Based Connection Semantics for Equation-Based Object-Oriented Modeling Languages

David Broman[1] and Henrik Nilsson[2]

[1] Department of Computer and Information Science, Linköping University, Sweden
david.broman@liu.se
[2] School of Computer Science, University of Nottingham, United Kingdom
nhn@cs.nott.ac.uk

Abstract. Declarative, Equation-Based Object-Oriented (EOO) modeling languages, like Modelica, support modeling of physical systems by composition of reusable component models. An important application area is modeling of cyber-physical systems. EOO languages typically feature a connection construct allowing component models to be assembled into systems much like physical components are. Different designs are possible. This paper introduces, formalizes, and validates an approach based on *explicit nodes* that expressly is designed to work for *functional* EOO languages supporting *higher-order modeling*. The paper also considers Modelica-style connections and explains why that design does not work for functional EOO languages, thus mapping out the design space.

Keywords: Declarative Languages, Modeling, and Simulation.

1 Introduction

Equation-based Object-Oriented (EOO) languages is an emerging class of declarative Domain-Specific Languages (DSLs) for modeling the dynamic aspects of systems using (primarily) differential equations [6]. These languages are characterized by *acausal* modeling of individual objects in the domain(s) of interest and composition of such object models into a complete system model[1]. Acausal modeling means there is no a priori assumption about the directionality of equations (known vs. unknown variables). This greatly facilitates reuse and composition [10], a crucial advantage for large models that can consist of thousands of equations. Moreover, EOO languages are typically capable of expressing models from arbitrary physical domains (e.g., mechanical, electrical, hydraulic) and of supporting *hybrid modeling*: modeling of both continuous-time and discrete-time aspects. State-of-the-art EOO languages include *Modelica* [11,19], VHDL-AMS [15]

[1] Some of these languages share typical traits of object-oriented programming languages, such as a class system, but this is not essential: object-oriented here refers to the focus on composition of reusable *models* that have a direct correspondence to *objects* in the *physical world*. Also, note that, unlike (imperative) object-oriented programming languages, EOO languages have no notion of mutable state.

C. Russo and N.-F. Zhou (Eds.): PADL 2012, LNCS 7149, pp. 258–272, 2012.

and Verilog-AMS [1]. Taken together, the characteristics of EOO languages make them particularly suitable for modeling Cyber-Physical Systems: complex systems that combine embedded computers and networks (the cyber) with physical processes [17]. Examples include cars, aircraft, and power plants.

Most EOO languages provide a mechanism to *connect* component models together in a way that mimics how physical components may be interconnected. To obtain a purely mathematical model, these connections have to be translated into equations. This translation is the *connection semantics*. Unsurprisingly, the connection semantics is grounded in physical reality, such as the conservation principles of various physical domains. Because these principles share a common mathematical structure, it is possible to formulate the connection semantics in a *domain-neutral* way. To that end, two kinds of physical quantities are distinguished: *flow* quantities and *potential* quantities. Connected flow quantities are translated into sum-to-zero equations, as a connection point itself does not provide any capability of storing the flowing quantity, while connected potential quantities are translated into equality constraints, as there can only be one potential at a connection point. Modelica is one language taking this approach.

While state-of-the-art EOO languages like Modelica are highly successful, they do have acknowledged weaknesses, including limited support for structurally dynamic systems and limited meta-modeling capabilities per se [20,26]. These and other considerations have led researchers to investigate a different approach to EOO language design that supports *higher-order* modeling. The common idea is to make models *first class* entities in the setting of a *functional language* and using pure functions as the central abstraction mechanism [6,14,20].

Unfortunately, the connection semantics of Modelica-like languages is predicated on specific design aspects of such languages and does not readily carry over to a functional setting with first-class models. Moreover, at least the Modelica connection semantics is complex and has not been fully formalized, making it difficult to understand it precisely (for end users as well as for implementors).

In this paper we propose an alternative approach to specifying the connection semantics based on *explicit* connection points, from now on *nodes*. The idea of explicit nodes is not new; for example, it is used in VHDL-AMS, Verilog-AMS, and other hardware description languages. The novel insight demonstrated in this paper is how a node-based approach solves the problem of defining the connection semantics in *functional* EOO languages. The resulting semantics is also pleasingly clear. In more detail, our specific contributions are:

- We relate Modelica-style connection semantics (Section 2) and the node-based approach (Section 3), thus mapping out part of the design space, and we explain why the former approach does not work in a functional setting.
- We formalize the semantics of the node-based approach (Section 4).
- We describe and validate a prototype implementation of the node-based approach in the *Modeling Kernel Language (MKL)* [6] (Section 5). (Note that MKL is just a vehicle: the approach as such is language-independent.)

2 Modelica-Style Approach

This section gives an informal overview of Modelica-style connection semantics and explains why this approach does not work in a functional setting. Our examples are from the analog electrical domain. However, we re-iterate that connection semantics in this paper is domain-neutral unless stated otherwise [8].

2.1 Models and Equation Generation

Fig. 1(a) depicts a graphical *model* of a simple electrical circuit. The model consists of five *component models*, in this case a voltage source `VS`, a resistor `R`, a capacitor `C`, an inductor `L`, and a ground `G`. At the lowest level of abstraction, a model consists of a set of *Differential-Algebraic Equations (DAEs)* [16]. For example, the behavior of resistor `R` is expressed declaratively by the algebraic equation `R*i = v` (Ohm's law) and the inductor's behavior is stated using the differential equation `L*der(i) = v`, where `der(i)` is the time derivative of `i`.

Each component model has one or more *ports* (or *connectors*) specifying its connection points. For example, the negative ports (white boxes) of the capacitor `C` and the inductor `L` are connected to the positive port (black box) of resistor `R`. In the analog electrical domain, each port has two variable instances, a *potential* variable `v` and a *flow* variable `i`, representing voltage and current respectively.

The connection semantics specifies how a *set of connected ports* is translated into equations over their instance variables. Two kinds of equations are generated: pairwise equalities among the potential variables, and a sum-to-zero equation for the flow variables. We use Modelica's dot-notation to refer to variables; e.g, `C.n.v` refers to `v` of the negative port `n` of the capacitor `C`. As an example, the port set {`G.p`, `R.n`, `VS.n`} (node a_3) is translated into the two equations `R.n.v = G.p.v` and `VS.n.v = G.p.v` for the potential variables and the sum-to-zero equation `G.p.i + R.n.i + VS.n.i = 0` for the flow variables.

2.2 Abstraction and Composition

In an EOO language, such as Modelica, a model is fundamentally a DAE system. However, to promote reuse and facilitate construction, models are usually

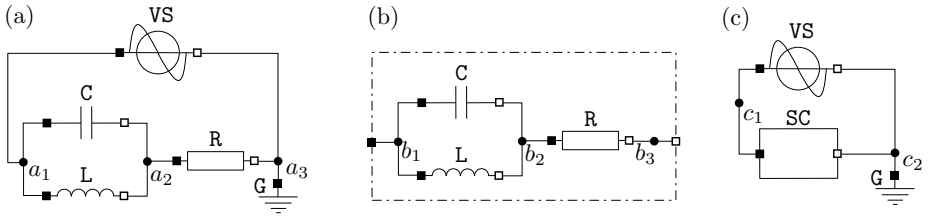

Fig. 1. Example of how parts of a circuit can be composed into a new model abstraction. Figure (a) shows the full circuit and (b) shows how three of the components are composed into a new model. Figure (c) shows how the model in (b) is used.

constructed hierarchically: related equations are grouped into models of physical components; such models can then be instantiated any number of times and further grouped into models of systems at progressively higher levels of abstraction.

For example, the model in Fig. 1(b) represents an abstraction of the components `R`, `C`, and `L` from Fig. 1(a). The dashed box represents the outside border of the abstracted model. Fig. 1(c) shows another way to model the circuit in (a), this time as a *composed* model using the sub-circuit in (b) (named `SC`) as one of the components. Hence, (a) and (c) model the exact same system, the only difference being that (c) introduces one more hierarchical level of abstraction.

The question is how to define connection semantics for composed models with several hierarchical levels of abstraction. In the Modelica-style, each port is considered either an *outside* or an *inside* port, depending on whether the current viewpoint is inside or outside a model. For example, in Fig. 1(b), when generating the sum-to-zero equation for the connection b_3, `SC.n` is considered an outside port and `SC.R.n` an inside port. The Modelica specification [19] states that outside connectors shall have a negative sign in sum-to-zero equations. The sum-to-zero equation at node b_3 is thus `-SC.n.i + SC.R.n.i = 0`. On the other hand, in model (c), port `SC.n` is considered an inside port, hence the resulting sum-to-zero equation for c_2 is `VS.n.i + SC.n.i + G.p.i = 0`. Information about the hierarchical structure is thus exploited when generating the equations.

2.3 Problems in a Functional Setting

In the Modelica-style approach, models have ports that define instance variables. A port is a *part* of the model it belongs to, and as such, its position in a compositional hierarchy becomes unambiguously determined; in particular, each port can be classified as inside or outside with respect to a specific model context and then treated accordingly for connection purposes.

In contrast, a functional EOO language uses *function abstraction* (or some variant thereof) for expressing model abstractions, with "ports" becoming *formal parameters*. As a result, a port is no longer *per se* a part with an implied position that can inform the generation of sum-to-zero equations. We can attempt to overcome this by introducing connection nodes as an independent notion. A model abstraction is then seen as a function mapping nodes to equations. But a node is just a node, a value like any other, without any special relation to specific abstractions, meaning that the notions inside and outside become meaningless. For example, assume that the model `SC` is defined as a function with two formal parameters. A function call `SC(c1,c2)` results in the nodes `c1` and `c2` being substituted into the function body of `SC`, yielding a collapsed hierarchy without any possibility to say whether a port is inside or outside.

Thus, the Modelica-style connection semantics does not carry over to a functional setting essentially because it is predicated on exploiting contextual information alien to this setting. To address this, we develop in the following an alternative approach that *is* suitable, based on nodes *and branches* (Electrical Engineering terminology; here essentially a directed edge annotated with variables) forming an explicit graph. Other possibilities are discussed in Sec. 6.3.

3 Node-Based Approach

This section informally describes the node-based approach to connection semantics. It has two phases: (1) Collapsing the hierarchical model structure into a directed graph of nodes, branches, and equations; (2) Translation of nodes and branches into additional equations, yielding a pure system of equations; i.e., the *connection semantics* proper. The approach is demonstrated using a small research language called the *Modeling Kernel Language (MKL)* [6]: a typed functional language specifically designed for embedding equation-based DSLs. However, note that the approach as such is language-independent.

3.1 Phase 1: Collapsing the Model Hierarchy

In an functional EOO-language, functions are used as the abstraction mechanism for describing composed models. For example, consider the following MKL model, which is the textual representation of Fig. 1(a):

```
def CircuitA() = {
    def a1,a2,a3:Electrical;
    SineVoltage(220,50,a1,a3);
    Capacitor(0.02,a1,a2);
    Inductor(0.1,a1,a2);
    Resistor(200,a2,a3);
    Ground(a3);
}
```

The model `CircuitA` is defined as a function without parameters. Three nodes `a1`, `a2`, and `a3` of type `Electrical` are defined. The five component models of the circuit are instantiated using function application; e.g., the application `Capacitor(0.02,a1,a2)` instantiates a capacitor of 0.02 F. The connection topology is defined by supplying the electrical nodes to the components; e.g., `Capacitor` is applied to nodes `a1` and `a2`. Note how both parallel and serial connections are expressed in this way (cf. Fig. 1(a)). The `Capacitor` model

```
def Capacitor(C:Real,p:Electrical,n:Electrical) = {
    def i:Current;
    def v:Voltage;
    Branch(i,v,p,n);
    C * der(v) = i;
}
```

has parameters `C` (capacitance) `p` (positive port), and `n` (negative port). Two unknown continuous-time signals `i` (current) and `v` (voltage) are defined inside the body. The third line in the body instantiates a `Branch` with four elements. Conceptually, a *branch* is a path between two nodes through a component model. Branches are essential for the translational connection semantics because they capture information necessary to generate correct signs in sum-to-zero equations.

Fig. 2 shows the resulting graph from evaluating the expression `CircuitA()`. Filled black arrows represent the branches (labeled edges). The nodes `a1`, `a2`,

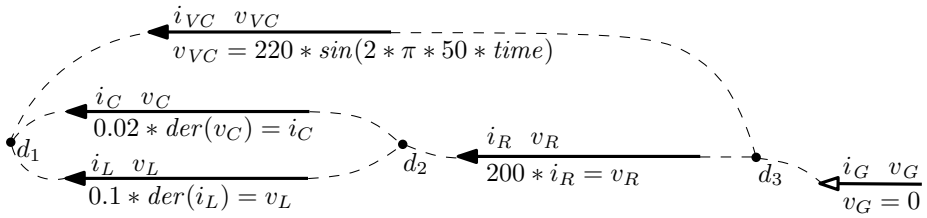

Fig. 2. The connection graph after collapsing the model hierarchy of `CircuitA` or `CircuitC`

and `a3` maps to d_1, d_2, and d_3 respectively. The graph is *directed* where the arrow head represents the positive position (the third element of a `branch`-instantiation) and the tail the negative position (forth element). The unknowns for a specific component are listed above each arrow. For example, i_R is the current flowing *through* the resistor branch and v_R is the voltage drop *across* the branch. The behavior equation for a specific component model is given below the arrow; e.g., Ohm's law in the resistor case. The unfilled arrow represents a *reference branch* (`RefBranch`) as used in the `Ground` model, for example:

```
def Ground(p:Electrical) = {
    def i:Current;
    def v:Voltage;
    RefBranch(i,v,p);
    v = 0;
}
```

Note that the `RefBranch` is only connected to one node. The intuition is that a reference branch makes the *absolute values* for a *specific node* accessible; i.e., the absolute potential value in relation to a global implicit reference value. The ground model states that the potential in the ground node is zero (`v = 0`).

So far we have only used basic components, such as `Resistor` and `Capacitor`. We now consider a model where one of the components itself is a composite model. The following is an MKL model of the sub-circuit from Fig. 1(b):

```
def SubCircuit(p:Electrical,n:Electrical) = {
    def b2:Electrical;
    Capacitor(0.02,p,b2);
    Inductor(0.1,p,b2);
    Resistor(200,b2,n);
}
```

The `SubCircuit` model is a function with two parameters `p` and `n`, both of type `Electrical`. A minor difference compared with Fig. 1(b) is that only node `b2` is defined inside the model: because a user of `SubCircuit` will supply the nodes between which it is going to be connected via parameters `p` and `n` (nodes being first-class), those nodes should *not* be defined inside `SubCircuit`. The model

```
def CircuitC(SC:TwoPin) = {
    def c1,c2:Electrical;
    SineVoltage(220,50,c1,c2);
    SC(c1,c2);
    Ground(c2);
}
```

is the MKL version of Fig. 1(c). It has one parameter `SC` of type `TwoPin`. This is an example where *Higher-Order Acausal Models (HOAMs)* [7] is used, i.e., where a model is parametrized with another model. The type `TwoPin`,

```
type TwoPin = Electrical -> Electrical -> Equations
```

is defined as a curried function[2] from nodes (type `Electrical`) to a system of equations (type `Equations`). Because `SubCircuit` is of type `TwoPin`, the expression `CircuitC(SubCircuit)` is well-typed and evaluating it results in a connection graph. During evaluation, `SC` is replaced with `SubCircuit`, meaning `SubCircuit` gets applied to the nodes `c1` and `c2`. Hence `c1` and `c2` are substituted for the formal parameters `p` and `n` respectively. The resulting connection graph for `CircuitC(SubCircuit)` is the same as that for Fig. 1(a), up to renaming of nodes. Thus, for `CircuitA()` the following holds: $d_1 = a_1$, $d_2 = a_2$, and $d_3 = a_3$, while for `CircuitC(SubCircuit)`: $d_1 = c_1$, $d_2 = b_2$, and $d_3 = c_2$.

3.2 Phase 2: The Connection Semantics

In the second phase, we translate the connection graph into a set of equations. We describe this translation process by defining three translation rules.

In contrast to the Modelica semantics, ports do not define instance variables. Nodes are instead defined explicitly in the model (e.g., d_1, d_2, and d_3 in Fig. 2), with each node corresponding to a *set* of connected ports in the Modelica approach. Instead of enforcing the equality of all potential variables of a port set by generating equality constraints, we apply the following rule:

> **Rule 1 - Potential variables:** Associate a distinct variable with each node in the system representing the potential in that node.

Three new distinct continuous-time variables v_{p1}, v_{p2}, and v_{p3} are thus associated with nodes d_1, d_2, and d_3 respectively.

A sum-to-zero equation must be created for each node and the signs in the equation must be chosen appropriately. This is where the information captured by *branches* comes into play. Consider the definition of `Capacitor` again. The first argument to `Branch` is the *flow variable* representing the current `i` through the branch, the second argument the *relative potential variable* representing the voltage `v` across the branch, the third argument the positive node `p`, and the fourth argument the negative node `n`. We can now define the second rule:

[2] All functions are curried in MKL even though the syntax of function definitions and applications uses parentheses. This design choice was made to make the functional style of programming more familiar to engineers used to the syntax of main-stream programming and modeling languages.

> **Rule 2 - Sum-to-zero equations:** For each node n in the circuit, create a sum-to-zero equation, such that the flow variables for the branches connected to node n get a positive sign if the branch is pointing towards the node, and a negative sign if it is pointing away from the node. For reference branches, the positive sign is always used.

Rule 2 results in the sum-to-zero equations $i_{VC} + i_C + i_L = 0$, $i_R - i_C - i_L = 0$, and $i_G - i_R - i_{VC} = 0$ for nodes d_1, d_2, and d_3 respectively.

The last translation rule defines the voltage across components:

> **Rule 3 - Branch equations:** For each branch in the model, create an equation stating that the *relative potential* across a branch is equal to the difference between the potential variable of the positive node and the one of the negative node. For a reference branch the relative potential is equal to the potential variable of the associated node.

Rule 3 results in one equation for each component; i.e., $v_{VC} = v_{p1} - v_{p3}$, $v_C = v_{p1} - v_{p2}$, $v_L = v_{p1} - v_{p2}$, $v_R = v_{p2} - v_{p3}$, and $v_G = v_{p3}$.

In the example, there are 13 variables in total: 10 variables originate from the potential and flow variables of each component, while 3 are generated from the nodes by rule 1. 5 behavior equations are explicitly stated for the model, 3 further equations are generated by rule 2 (sum-to-zero), and 5 more by rule 3. There are thus 13 equations and 13 variables: a necessary but not sufficient condition for solving a set of independent equations.

We note the following invariants. First, for each node, rule 1 adds one variable and rule 2 adds one equation. Second, two variables are always defined for each component: one flow variable and one relative potential variable. There are also always two equations for each component: one behavior equation defined in the original component model, and one branch equation generated by rule 3.

These invariants make it clear that the balance between the number of variables and equations is preserved under interconnection of correctly defined components. The approach is thus correct in that sense. However, the number of generated equations is not minimal; for example, a sum-to-zero equation can always be eliminated by using it to solve for one variable and substitute the result into other equations. However, we are not concerned with such issues here as that has to do with solving the equations, not with the semantics of connections.

4 Formalization of the Connection Semantics

In this section we formalize the node-based connection semantics. Note that the formalization is independent of MKL.

4.1 Notation and Syntax

Let N be a finite set of nodes and $n \in N$ denote a node element. Let V be a finite set of variables and $v \in V$ a variable. Let B_{bin} be the set of binary

branches and B_{ref} be the set of unary reference branches. A binary branch is a quadruple $(v_f, v_{rp}, n_1, n_2) \in B_{bin}$, where v_f is a flow variable, v_{rp} a relative potential variable, n_1 a first and n_2 a second node connected to the branch. A reference branch is a triple $(v_f, v_{rp}, n_1) \in B_{ref}$, where v_f is the flow variable, v_{rp} a relative potential variable, n_1 a connected node. Let $B = B_{bin} \cup B_{ref}$ be the set of all branches. The syntax of expressions e is given by the grammar rules

$$e ::= e \texttt{ + } e \mid e \texttt{ - } e \mid \texttt{0} \mid v$$

where `+` and `-` are the plus and minus operators, `0` the value zero, and v a variable. The syntax for an equation is $e_1 \texttt{ = } e_2$, where e_1 and e_2 are expressions. Let E be a multiset of equations. A multiset is needed as equations could be repeated in a model[3].

We use braces to denote sets and square brackets to denote multisets. When pattern matching on sets, the pattern $A \cup \{a\}$ matches a non-empty set with a being bound to an arbitrary element of the set and A being bound to the rest of the set, *not* including a.

We postulate an overloaded function *vars* that returns the set of variables occurring in a branch, an expression, or a (multi)set of branches or expressions. Similarly, we postulate an overloaded function *nodes* that returns the set of nodes occurring in a branch or set of branches.

4.2 Semantics of Rules

Fig. 3 defines the connection semantics using (recursive) function definitions. The functions are categorized according to the informal rules in previous section.

Rule 1 associates a new potential variable with each node. The function *potvar* returns a bijective function *pv* mapping each node to a corresponding potential variable, distinct from any of the existing variables V_{BE}.

Rule 2 describes the generation of the multiset of sum-to-zero equations. The rule defines one main function *sumzeroeqns* and one auxiliary function *sumexpr*. The function *sumezeroeqns* takes two arguments, where the first argument N is the set of nodes and the second argument B the set of branches. For each $n \in N$, the function creates the corresponding sum-to-zero expression using set-builder notation for multisets together with calling *sumexpr*. The first three cases of *sumexpr* concern binary branches by matching on the quadruple (v_f, v_{rp}, n_1, n_2). Only branches directly connected to the node under consideration contribute to the expression. The last two cases handle reference branches in the same manner. Note that a literal 0 is inserted at the end of the recursion. This zero could easily be eliminated by introducing unary minus in the expression syntax. However, this would make the formalization less readable.

Rule 3 describes the generation of the multiset of relative potential equations. The rule defines a function *brancheqns* that takes two arguments. The first argument *pv* is the mapping between nodes and potential variables (see Rule 1).

[3] We do not wish to eliminate redundant equations here, and we note that syntactic equality on equations would not suffice for this purpose anyway.

The second argument B is the set of branches. Different equations are generated depending on whether a branch is a binary branch or a reference branch.

The last function definition *consem* takes the set B of branches and multiset E of equations that already exists in the model (i.e, the behavior equations) as arguments. The function returns the final multiset of model equations; i.e., the initial equations along with all generated equations.

A branch starting and ending at the same node is a bit of a special case. The relative potential across such a branch is, of course, 0, and no special consideration is needed in rule 3 for the associated potential variable. However, such a branch *in itself* imposes *no* constraints on the flow through it. Rule 2 thus

Rule 1 - Potential variables $\boxed{potvar(N, V_{BE})}$

$$potvar(N, V_{BE}) = pv \quad \text{where} \quad pv : N \to V_P \text{ is bijective, } V_P \subseteq V, \text{ and } V_P \cap V_{BE} = \emptyset$$

Rule 2 - Sum-to-zero equations $\boxed{sumzeroeqns(N, B)}$

$$sumzeroeqns(N, B) \;=\; [\; sumexpr(n, B) \texttt{ = 0} \mid n \in N \;]$$

$\boxed{sumexpr(n, B)}$

$$sumexpr(n, \emptyset) \;=\; \texttt{0}$$

$$sumexpr(n, B \cup \{b\}) \;=\; \begin{cases} sumexpr(n, B) \texttt{ + } v_f & \text{if } (v_f, v_{rp}, n_1, n_2) = b \text{ and } n = n_1 \text{ and } n \neq n_2 \\ sumexpr(n, B) \texttt{ - } v_f & \text{if } (v_f, v_{rp}, n_1, n_2) = b \text{ and } n \neq n_1 \text{ and } n = n_2 \\ sumexpr(n, B) & \text{if } (v_f, v_{rp}, n_1, n_2) = b \text{ and } ((n \neq n_1 \text{ and } n \neq n_2) \text{ or } (n = n_1 \text{ and } n = n_2)) \\ sumexpr(n, B) \texttt{ + } v_f & \text{if } (v_f, v_{rp}, n_1) = b \text{ and } n = n_1 \\ sumexpr(n, B) & \text{if } (v_f, v_{rp}, n_1) = b \text{ and } n \neq n_1 \end{cases}$$

Rule 3 - Branch equations $\boxed{brancheqns(pv, B)}$

$$brancheqns(pv, B) = [\; eqn(b) \mid b \in B \;] \quad \text{where}$$

$$eqn(b) = \begin{cases} v_{rp} \texttt{ = } pv(n_1) \texttt{ - } pv(n_2) & \text{if } b = (v_f, v_{rp}, n_1, n_2) \\ v_{rp} \texttt{ = } pv(n_1) & \text{if } b = (v_f, v_{rp}, n_1) \end{cases}$$

Translational connection semantics $\boxed{consem(B, E)}$

$$consem(B, E) = E \cup sumzeroeqns(N, B) \cup brancheqns(pv, B) \quad \text{where}$$

$$N = nodes(B)$$
$$V_{BE} = vars(B) \cup vars(E)$$
$$pv = potvar(N, V_{BE})$$

Fig. 3. Formalization of the node-based connection semantics

carefully ignores any such branch, meaning that the associated flow variable will not appear in any sum-to-zero equation. (Of course, it would usually appear in other equations, like component equations relating the relative potential and flow.)

5 Implementation and Evaluation

We have developed a prototype implementation of the node-base connection semantics as a functional EOO DSL in MKL. The prototype has three parts:

- Libraries for defining the elaboration semantics of a functional EOO DSL supporting acausal modeling in the continuous-time domain. The connection semantics that is part of the elaboration semantics was implemented according to the formalization presented in this paper, with certain optimizations together with more efficient data structures.
- Libraries for defining reusable components (models of physical objects) within the analog electrical domain, the rotational mechanical domain, and automatic control domain.
- Test models that use the modeling libraries.

The evaluation of the prototype so far was concerned with testing the correctness of the node-based approach compared to Modelica's approach. The selected test models were chosen according to the following criteria:

- Size of the model, where the largest model contained more than 1000 equations after translation.
- Combination of and interaction between different physical domains, like electrical, mechanical, and control, to ensure domain-neutrality.
- Modeling abstraction and generation mechanisms, such as higher-order models and recursively defined models.

The test procedure was as follows:

1. The model was created in Modelica using standard components in Modelica standard library.
2. The same model was created by using components from MKL's standard library. This library has been modeled according to the Modelica library.
3. The Modelica model was simulated using Dymola 6 [9], a Modelica environment. Data from the sensors was plotted and visualized.
4. The MKL model was translated into flat equations by the prototype implementation following the connection semantics defined in this paper. Dymola 6 was then used as a simulation backend to simulate and plot these flat equations. Using the same simulation backend for both the model expressed in Modelica and for the model expressed in MKL eliminates the risk of differences in the results due to differences in employed simulation methods.
5. The plotted results from the Modelica model and the MKL model were visually compared.

In all cases the simulation result from the Modelica models were found to coincide with the results from the corresponding MKL version of the model; i.e., the results were the same. This confirms the described approach works as intended, in a functional setting, and is applicable for multi-physical modeling. Moreover, preliminary performance measurements of the translational semantics show that the approach can scale up to hundreds of thousands equations. Our approach has not yet been evaluated for structurally dynamic systems, which we see as the next step of future work.

6 Related Work

6.1 Modelica

The work most closely related to the node-based approach is the connection semantics for Modelica [11,19]. As we saw (Sec. 2), Modelica lets the modeler specify sets of interconnected component ports. Each such set corresponds to a node and is translated into connection equations by taking the context-dependent classification of individual ports as being outside or inside into account. However, nodes are *not* an explicit notion. In contrast, to provide connection functionality without relying on specific language design aspects (beyond the standard notion of functions), nodes along with branches are made *explicit* notions in the node-based approach and used to construct an *explicit* interconnection graph containing all necessary information for subsequent translation into connection equations. This approach is thus a good fit for e.g. functional EOO languages as the kind of contextual information used in Modelica is not available (Sec. 2.3).

Furic [12] proposes an alternative connection semantics for Modelica. The main objective is to make models compose better and to support structural dynamism. For example, in Modelica, missing or "duplicated" ground references in electrical models typically lead to under- and over-constrained systems of equations respectively, and ideal switches might mean there is no one way of "grounding" the model that works for all structural configurations. Furic's approach is based on nodes, like our approach, but, following VHDL-AMS, it employs *relative* potentials across branches between nodes, referred to as *effort*, while *absolute* potentials at nodes are of no concern, unlike in our approach and the standard Modelica approach. The end result is an explicit representation of the model topology in the form of a graph, like in our case, which suggests that it may be possible to adapt Furic's approach to a functional setting. However, like for VHDL-AMS, special source and sensor constructs are necessary to mediate between the "effort/flow world" and the "signal world", e.g. to feed in external stimuli or make observations. This is more direct in our setting. Furic's work has not yet been formalized or thoroughly evaluated outside the electrical domain, but constitute another interesting node-based approach.

6.2 Hardware Description Languages

Hardware Description Languages, such as VHDL and Verilog, are primarily used for describing digital electrical circuits. However, there exist *analog and mixed*

signal (AMS) extensions to both these languages: VHDL-AMS [2] and Verilog-AMS [1] respectively. These variants allow modeling of continuous systems from various physical domains. Both VHDL-AMS and Verilog-AMS have a node-based connection semantics, where nodes connect components together via ports. However, in contrast to the work presented in this paper, neither language has a *formally* specified semantics for connections. The VHDL-AMS specification [15] describes the connection semantics informally as part of the elaboration phase of the language. Similarly, Verilog-AMS definition states that DAE equations are generated according to Kirchhoff's laws, but does not specify how.

Lava [4] is a tool for specifying and verifying hardware circuits. It is embedded in Haskell and makes use of higher-order functions and combinators for composing circuits. Wired [3] is a relational language that is based on Lava, but also models the layout of a circuit, including the wires. Both Lava and Wired are used for describing *digital* circuits; the kind of connections discussed here grounded in abstraction over phenomena from continuous physics is thus not relevant. However, both employ a notion of explicit nodes for describing circuits.

SPICE [23] is a circuit simulation program originally developed at UC Berkeley in the 1970s. Circuits are defined using *netlists*, a textual description where electrical components are connected together using nodes. SPICE uses a modified nodal analysis method with special treatment for voltage sources to enable numerical approximation. In contrast, our approach generates DAEs as output and relies on symbolic/numerical methods developed in the 1980s-1990s for solving DAEs [18,21,22]. Also, SPICE is designed for analog circuit simulation, whereas our approach is based on ideas from Modelica and is domain-neutral.

6.3 Functional Acausal Languages

The Flow λ-calculus [5] is a minimal EOO language developed by the first author. It is an extension of the λ-calculus with primitives for generating flow equations. The approach to connections taken by the Flow λ-calculus inspired the node-based approach presented here, but its semantics was more complex.

Functional Hybrid Modeling (FHM) [20] combines functional programming and acausal modeling. It can be seen as a generalization of causal Functional Reactive Programming (FRP) [25]. Hydra is a DSL within the FHM paradigm developed by Giorgidze and Nilsson [14]. At present, the language is realized as an embedding in Haskell [24], with just-in-time compilation of simulation code for speed. FHM supports highly structurally dynamic systems and it makes a strict distinction between time-invariant and time-varying entities, relegating the latter to secondary status. The central FHM modeling-specific abstraction is the *signal relation*. It is similar to model abstraction in MKL, but formally parametrized on *signals*, time-varying values, not nodes.

Modelica-style connections are not applicable to FHM for the reasons outlined in Sec. 2.3. Instead, a scheme is adopted with one `connect`-specification per node enumerating *all* variables related by that node [13]. By assuming that flow is always directed into a signal relation, the signs of the flow variables in the generated sum-to-zero equations are always positive, independent of context.

Signal relation *application* then takes care of the necessary sign-reversal for flow quantities (what flows into one signal relation, flows out of another).

While this scheme is simple and quite effective, it does require connections to be expressed in a particular way. For example, and perhaps unexpectedly, connection by transitivity does not work. While static checks can be employed to catch mistakes, the node-based approach would be an interesting alternative.

7 Conclusions

We presented and formalized a new, node-based approach to specifying model composition through connections in the context of equation-based, acausal languages for modeling of physical systems. The main benefit compared to the connect-based approach used in Modelica is that it does not assume much about the language design. Thus it works well for, for example, functional EOO languages, which, indeed, was the goal of the design. Additional advantages include its simplicity and clarity, as evidenced by the formalization.

Acknowledgements. The authors would like to thank Peter Fritzson and John Capper for useful comments. The first author was funded by the ELLIIT project.

References

1. Accellera Organization. Verilog-AMS Language Reference Manual - Analog & Mixed-Signal Extensions to Verilog HDL Version 2.3.1 (2009)
2. Ashenden, P.J., Peterson, G.D., Teegarden, D.A.: The System Designer's Guide to VHDL-AMS: Analog, Mixed-Signal, and Mixed-Technology Modeling. Morgan Kaufmann Publishers, USA (2002)
3. Axelsson, E., Claessen, K., Sheeran, M.: Wired: Wire-Aware Circuit Design. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 5–19. Springer, Heidelberg (2005)
4. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in Haskell. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, pp. 174–184. ACM Press, New York (1998)
5. Broman, D.: Flow Lambda Calculus for Declarative Physical Connection Semantics. Technical Reports in Computer and Information Science No. 1. LiU Electronic Press (2007)
6. Broman, D.: Meta-Languages and Semantics for Equation-Based Modeling and Simulation. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden (2010)
7. Broman, D., Fritzson, P.: Higher-Order Acausal Models. Simulation News Europe 19(1), 5–16 (2009)
8. Cellier, F.E.: Continuous System Modeling. Springer, New York (1991)
9. Dassault Systems. Multi-Engineering Modeling and Simulation - Dymola - CATIA - Dassault Systemes, `http://www.dymola.com` (last accessed: September 16, 2011)
10. Elmqvist, H., Mattsson, S.E., Otter, M.: Modelica - A Language for Physical System Modeling, Visualization and Interaction. In: Proceedings of the IEEE International Symposium on Computer Aided Control System Design (1999)

11. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press, New York (2004)
12. Furic, S.: Enforcing model composability in Modelica. In: Proceedings of the 7th International Modelica Conference, Como, Italy, pp. 868–879 (2009)
13. Giorgidze, G., Nilsson, H.: Embedding a Functional Hybrid Modelling Language in Haskell. In: Scholz, S.-B., Chitil, O. (eds.) IFL 2008. LNCS, vol. 5836, pp. 138–155. Springer, Heidelberg (2011)
14. Giorgidze, G., Nilsson, H.: Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems. In: Proceedings of the 7th International Modelica Conference, Como, Italy, pp. 208–218. LiU Electronic Press (September 2009)
15. IEEE Std 1076.1-2007. IEEE Standard VHDL Analog and Mixed-Signal Extensions. IEEE Press (2007)
16. Kunkel, P., Mehrmann, V.: Differential-Algebraic Equations Analysis and Numerical Solution. European Mathematical Society (2006)
17. Lee, E.A.: CPS foundations. In: Proceedings of the 47th Design Automation Conference, DAC 2010, pp. 737–742. ACM Press, New York (2010)
18. Mattsson, S.E., Söderlind, G.: Index reduction in differential-algebraic equations using dummy derivatives. SIAM Journal on Scientific Computing 14(3), 677–692 (1993)
19. Modelica Association. Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.2 (2010), `http://www.modelica.org`
20. Nilsson, H., Peterson, J., Hudak, P.: Functional Hybrid Modeling. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 376–390. Springer, Heidelberg (2002)
21. Pantelides, C.C.: The Consistent Initialization of Differential-Algebraic Systems. SIAM Journal on Scientific and Statistical Computing 9(2), 213–231 (1988)
22. Petzold, L.R.: A Description of DASSL: A Differential/Algebraic System Solver. In: IMACS Trans. on Scientific Comp., 10th IMACS World Congress on Systems Simulation and Scientific Comp., Montreal, Canada (1982)
23. Quarles, T.L., Newton, A.R., Pedersen, D.O., Sangiovanni-Vincentelli, A.: SPICE3 Version 3f3 User's Manual. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley (1993)
24. Jones, S.P.: Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press (2003)
25. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: PLDI 2000: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 242–252. ACM Press, New York (2000)
26. Zimmer, D.: Enhancing Modelica towards variable structure systems. In: Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, Berlin, Germany, pp. 61–70. LiU Electronic Press (2007)