Technical reports in Computer and Information Science

Report number 1.

Flow Lambda Calculus for Declarative Physical Connection Semantics

by

David Broman
davbr@ida.liu.se

December 17, 2007

Linköping University
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Technical reports in Computer and Information Science are available online at
Linköping Electronic Press: http://www.ep.liu.se/ea/trcis/

Flow Lambda Calculus for Declarative Physical Connection Semantics

Technical Reports in Computer and Information Science. No. 1.
Linköping University Electronic Press

David Broman
Department of Computer and Information Science
Linköping University, SE–581 83 Linköping, Sweden
davbr@ida.liu.se

December 17, 2007

Abstract

One of the most fundamental language constructs of equation-based object-oriented languages is the possibility to state acausal connections, where both potential variables and flow variables exist. Several of the state-of-the art languages in this category are informally specified using natural language. This can make the languages hard to interpret, reason about, and disable the possibility to guarantee the absence of certain errors. In this work, we construct a formal operational small-step semantics based on the lambda-calculus. The calculus is then extended with more convenient modeling capabilities. Examples are given that demonstrate the expressiveness of the language, and some tests are made to verify the correctness of the semantics.

Keywords: Flow connection, Flow Lambda Calculus, Operational Semantics

1 Introduction

Modeling and simulation have been an important application area for several successful programming languages, e.g., Simula [4] and C++ [12]. These languages and other general-purpose languages can be used efficiently for discrete time/event-based simulation, but for continuous-time simulation, other specialized tools such as Simulink [8] are commonly used in industry. The latter supports causal block-oriented modeling, where each block has defined input(s) and output(s). However, during the last decades, a new kind of language has emerged, where differential algebraic equations (DAEs) can describe the continuous-time behaviour of a system. These languages enable modeling of complex physical systems by combining different domains, such as electrical, mechanical, and hydraulic. Examples of such a languages are Modelica [9], Omola [1], gPROMS [2, 11], VHDL-AMS [3], and χ (Chi) [5, 13]. Several of

these languages (e.g., Modelica and Omola) support object-oriented concepts, where physical models can be composed and reused. One of the fundamental concepts enabling this composition, is the use of acausal connections between model instances, with the use of *potential* and *flow* variables. These kinds of variables are common in most physical domains and describe the preservation of energy in a system. For example, in the electrical domain, potential variables denote voltage potential and flow variables denote electric current, which obey Kirchhoff's current law, i.e., that the current should sum to zero in a node. As another example, in the rotational mechanical domain, angles are expressed using potential variables and torque is represented using flow variables.

1.1 Motivation and Contribution

Languages of this sort have been developed from an engineering perspective with the focus on numerical solution strategies and run-time semantics for handling mixed discrete / continuous-time (hybrid) systems. Several of these languages have grown to be large and are informally specified using natural language. This can make the languages hard to interpret, maintain, and reason about, which affects both tool development and language evolution. Moreover, the need for static detection and isolation of certain modeling errors is essential for productive modeling and simulation. Such errors can concern over- and under-constrained systems of equations, and consistency checking of physical units and dimensions. Even if current tools support checking for these kind of errors, a formal semantics of the language is needed to be able to develop checking algorithms that *guarantee* the absence of faults.

Hence, there is a concrete need to be able to express the core concepts of such equation-based object-oriented (EOO) languages using formal semantics. We have in this paper developed a novel small-step operational semantics that captures the essential constructs in such languages, including acausal connections, potential and flow-variables, and model abstraction. The semantics is built on the untyped λ-calculus, which is extended with semantics for handling flow-connections.

1.2 Outline

The remainder of this paper is structured as follows. Section 2 gives an informal introduction to acausal physical modeling using the concept of higher-order models and functional abstraction. An example of a simple circuit is modeled and the concept for model reuse and specialization is outlined. Section 3 states the formal abstract syntax and operational semantics of the untyped flow λ-calculus (written $\tilde{\lambda}$). This syntax and semantics forms the basis of the modeling kernel language (MKL), which is presented in Section 4. The additional formal syntax and formal semantic rules are given and syntactic derived forms are described. The language presented in this section is the one used in the modeling examples in Section 2. Section 5 describes the prototype implementation and gives a short evaluation of the language semantics and Section 6 presents related work. Finally, Section 7 states concluding remarks.

2 Informal Language Syntax and Semantics

In this section, an informal introduction to an experimental language called Modeling Kernel Language (MKL) is outlined. The language is not intended to be a full fledged modeling language, but to demonstrate the fundamental modeling possibilities when a equation-based modeling language is based on the lambda calculus.

2.1 A Simple Electrical Circuit

To illustrate the basic modeling capabilities, the simple circuit shown in Figure 1 is to be modeled and simulated.

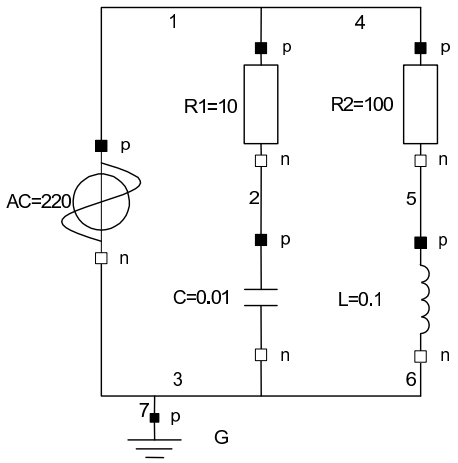


Figure 1: Graphical outline of a simple electrical circuit.

The model is described by the following source code:

```
def Circuit = model()
{
    def w1 = Wire();
    def w2 = Wire();
    def w3 = Wire();
    def w4 = Wire();
    Resistor(w1,w2,10);
    Capacitor(w2,w4,0.01);
    Resistor(w1,w3,100);
    Inductor(w3,w4,0.1);
    VSourceAC(w1,w4,220);
    Ground(w4);
};
```

The code shows the definition of a new model called `Circuit`. The model takes zero formal parameters, given by the empty tuple to the right of the model keyword, `model()`. The content of the model is given within curly braces. The first four statements define four new *wires*, e.g., connection points from which the different components (model instances) can be connected.

The six components defined in this circuit correspond to the layout given in Figure 1. Consider the first resistor instantiated using the following:

```
Resistor(w1,w2,10);
```

The two first arguments state that wires `w1` and `w2` will be connected to this resistor. The last argument expresses that the resistance for this instance will be 10 Ohm. Wire `w2` is also given as argument to the capacitor, stating that the first resistor and the capacitor are connected using wire `w2`.

2.2 Connections, Variables, and Flow Nodes

The concept of wire is not built into the language. Instead, it is defined as follows:

```
def Wire = func(){(var(),flow())};
```

Here, a function called `Wire` is defined by using the anonymous function construct `func`. The definition states that function takes an empty tuple `()` as argument and returns the expression within curly braces. In this case, a tuple `(var(),flow())` with two elements is returned. A tuple is expressed as a sequence of terms separated by commas and enclosed in parentheses.

The first element of the defined tuple expresses the creation of a new unknown continuous-time variable using the syntax `var()`. The variable could have been given an initial value, which is used as a start value when solving the differential equation system. For example, creating a variable with initial value 10 can be written using the expression `var(10)`. Variables defined using `var()` correspond to *potential* variables, i.e., the voltage in this example.

The second part of the tuple expresses the current in the wire by using the construct `flow()`, which creates a new flow-node. This construct is the essential part in the semantics presented in coming sections. In this informal introduction, we just accept the fact that Kirchhoff's current law with sum to zero at nodes is managed in a correct way.

In the circuit definition we used the syntax `Wire()`, which means that the empty tuple `()` is supplied as a tuple argument to the function `Wire`. The function call will return the tuple `(var(),flow())`. Hence, the `Wire` definition is used for encapsulating the tuple, allowing the definition to be reused without the need to to restate its definition over and over again.

2.3 Models and Equation Systems

The main model in this example is already given as the `Circuit` model. This model contains instances of other models, such as the `Resistor`. These models are also defined using model definitions. Consider the following two models:

```
def TwoPin = model((pv,pi),(nv,ni),v)
{
    v = pv - nv;
    0 = pi + ni;
};

def Resistor = model(p,n,R)
{
    def (_,pi) = p;
    def v = var();
    TwoPin(p,n,v);
    R*pi=v;
};
```

Models are defined anonymously using the keyword `model` followed by a formal parameter and the model's content stated within curly braces. The formal parameter can be a pattern and *pattern matching* is used for decomposing arguments. Inside the body of the model, definitions, components, and equations can be stated in any order within the same scope.

The general model `TwoPin` is used for defining common behavior of a model with two connection points. `Twopin` is defined using an anonymous model, which here takes one formal parameter. This parameter specifies that the argument must be a 3-tuple with the specified structure, where `pv`, `pi`, `nv`, `ni`, and `v` are pattern variables. Here `pv` means positive voltage, and `ni` negative current. Since the illustrated language is untyped, illegal patterns will be discovered first during run-time.

Both models contain new definitions and equations. The equation `v = pv - nv;` in `TwoPin` states the voltage drop over a component that is an instance of `TwoPin`. The definition of the voltage `v` is given as a formal parameter to `TwoPin`. Note that the direction of the causality of this formal parameter is not defined at modeling time.

The resistor is defined in a similar manner, where the third element `R` of the input parameter is the resistance. The first line `def (_,pi) = p;` is an alternative way of pattern matching where the current `pi` is extracted from `p`. The pattern _ states that the matched value is ignored. The second row defines a new variable `v` for the voltage. This variable is used both as an argument to the instantiation of `TwoPin` and as part of the equation `R*pi=v;` stating Ohm's law. Note that the wires `p` and `n` are connected directly to the `TwoPin` instance.

The capacitor and inductor models are defined as follows:

```
def Capacitor = model(p,n,C)
{
    def (_,pi) = p;
    def v = var(0);
    TwoPin(p,n,v);
    C*der(v) = pi;
};

def Inductor = model(p,n,L)
{
    def (_,pi) = p;
    def v = var(0);
    TwoPin(p,n,v);
    L*der(pi) = v;
};
```

It should be noted here that each of these models contains a differential equation. For example in equation `L*der(pi) = v;`, the `pi` variable is differentiated with respect to time using the built-in `der` operation.

Finally, to make the example complete, the voltage source and the ground are defined as follows:

```
def VSourceAC = model(p,n,VA)
{
    def v = var(0);
    TwoPin(p,n,v);
    def f = 50;
    def PI = 3.14;
    v = VA*sin(2*PI*f*time);
};

def Ground = model((pv,_))
{
    pv = 0;
};
```

An instance of the `Circuit` model can be created in the top-level scope using the following code:

```
Circuit();
```

The resulting simulation result is shown in Figure 2.

2.4 Reuse and Expressiveness using Higher-Order Models

Models are very closely related to anonymous functions. We will see later that the models are in fact encoded as lambda abstractions, with special care taken to flow connections. Since models are first class citizens, reuse and expressive modeling can make use of *higher-order models*, i.e., models and functions can take models as arguments and return new models.

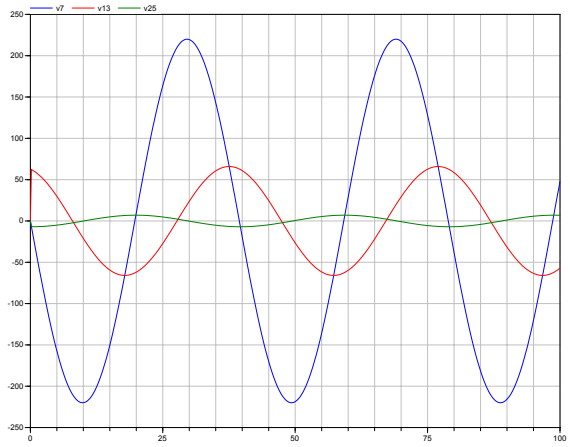


Figure 2: Plot of simulation result of the simple circuit. The largest curve shows the voltage source, the second largest the voltage drop over the inductor, and the smallest one the voltage drop over the capacitor.

For example, let us assume that we want to create a new model, which connects two Resistors in parallel. This model can be defined as follows, with resistance values 10 and 100:

```
def ParallelResistor = model(p,n)
{
    Resistor(p,n,10);
    Resistor(p,n,100);
};
```

This simple definition defines a new model named `ParallelResistor`, which composes two resistor instances. Hence, a new model can be defined by reusing other models in an hierarchical structure.

However, can we not generalize this and create a generic way for composing models? Assume that we want to create a model based on composing a few existing models in series. However, we do not want to do it from scratch, e.g., create a model where two resistors are composed in series and then yet another model for an inductor and a capacitor in series. Consider the following function, which takes two models `M1` and `M2` as input, plus an attribute value for the model, such as the resistance or the inductance.

```
def makeSerial = func(M1,val1,M2,val2){
    model(pin,pout){
        def w = Wire();
        M1(pin,w,val1);
        M2(w,pout,val2);
    }
};
```

The function `makeSerial` creates instances of the models `M1` and `M2`, and connects them together using the wire `w`. The left side of `M1` is connected to the new anonymous model's port `pin`, and the second port of `M2` is connected to `pout`. The generic function then returns this new model.

An example where this function is used is given in the following circuit:

```
def Circuit2 = model()
{
    def w1 = Wire();
    def w2 = Wire();
    def ResInd = makeSerial(Resistor, 100, Inductor, 0.1);
    def CapRes = makeSerial(Capacitor, 0.01, Resistor, 200);
    ResInd(w1,w2);
    CapRes(w1,w2);
    Ground(w2);
    VSourceAC(w1,w2,5);
};
```

Here, `makeSerial` defines a new model called `ResInd`, by composing a `Resistor` and an `Inductor`. In the same way model `CapRes` is defined, by composing a `Capacitor` and another `Resistor`.

Note that models can also be parameterized and specialized using traditional concepts in functional programming, e.g., by using currying.

3 Flow Lambda Calculus

In this section, the new connection semantics of flow variables is presented, by extending the untyped lambda calculus with a number of terms, values, and rules. We call this extended version of the lambda-calculus for *flow lambda-calculus*, denoted $\tilde{\lambda}$-calculus.

3.1 Abstract Syntax

Consider the abstract syntax of the $\tilde{\lambda}$-calculus listed in Figure 3. Besides the standard terms lambda abstraction, application, and identifier, a number of terms have been added.

The equation term t_1=t_2 expresses a differential or algebraic equation. The conjunction term $t_1 \wedge t_2$ is used for composing equations into a tree, forming an equation system.

The term `var(`t`)` constructs a new *variable location* (potential variable) in the *variable store*, σ. The creation of flow variables in this store is described in Section 3.2. The store consists of a mapping from a *variable store location* l to a value, σ : VLoc $\rightarrow_{\text{fin}}$ Value. When creating models with systems of equations, variables are often unknown before simulation. An unknown variable is a mapping from a variable store location l to the unknown term ϵ.

The most essential part in this calculus is the definition and treatment of flow nodes together with the flow store. During evaluation, the flow nodes are combined into a tree, which is stored in the flow store ϕ. This tree controls the sum to zero equations that are going to be part of the equation system. The flow store is a finite map from a *flow store location*, f, to a specific node in the tree. Nodes in the tree can be colored to be either black or white, which is represented in the abstract syntax by terminals $\mathcal{N}_{\mathcal{B}}(t, n)$ and $\mathcal{N}_{\mathcal{W}}(t, n)$. After evaluation, the black nodes represent the sum to zero equations. The white nodes are used during the construct of the tree, but are not representing any equations. Variables, sometimes referred to as *flow variables*, which are used in

r	$\in \mathbb{R}$	Real number
x	$\in$ Ident	Identifier
l	$\in$ VLoc	Variable Store location
f	$\in$ FLoc	Flow Store location
σ	$\in$ VStore $=$ VLoc $\rightarrow_{\text{fin}}$ Value	Variable Store
ϕ	$\in$ FStore $=$ FLoc $\rightarrow_{\text{fin}}$ FNode	Flow Store
n	$\in$ FNode	**Flow nodes**
	$n ::=$	
	$\mathcal{N}_{\mathcal{B}}(t, n)$	Black node
	$\mid \quad \mathcal{N}_{\mathcal{W}}(t, n)$	White node
	$\mid \quad \mathcal{N}_{\mathcal{E}}$	Empty node
t	$\in$ Term	**Terms**
	$t ::=$	
	$\lambda x.t$	Lambda abstraction
	$\mid \quad t_1 t_2$	Application
	$\mid \quad r$	Real number
	$\mid \quad x$	Identifier
	$\mid \quad t_1 = t_2$	Equation
	$\mid \quad t_1 \wedge t_2$	Conjunction
	$\mid \quad$ `var(`t`)`	Variable constructor
	$\mid \quad$ `flow()`	Flow node constructor
	$\mid \quad$ `fork(`t`)`	Fork connection
	$\mid \quad l$	Variable Store location
	$\mid \quad f$	Flow Store location
	$\mid \quad \epsilon$	Unknown
v	$\in$ Value	**Values**
v	$::=$	
	$\lambda x.t \mid r \mid v_1 = v_2$	
	$\mid \quad v_1 \wedge v_2 \mid l \mid f \mid \epsilon$	

Figure 3: Abstract Syntax for $\tilde{\lambda}$–calculus.

the sum to zero equations, are created in the variable store σ, and referred to in the flow nodes located in the flow store ϕ.

New nodes are added to the flow store by evaluation of the term `flow()`. Recall the definition of `Wire` in Section 2, which consisted of a tuple with terms `var(`t`)` and `flow()` as elements. A new flow node is created when this tuple is evaluated.

The last new term is `fork(`t`)`. This is the essential term used for flow connections. It is an internal term, which does not need to be created explicitly by the user of the language. Instead, this term can be hidden and created implicitly by other more convenient constructs. Section 4 describes in more detail how this simplification transformation is performed.

The described $\tilde{\lambda}$-calculus is defined using call-by-value evaluation order. Hence, the set Value $\subseteq$ Term, is used for determining when a term has been evaluated to a value.

3.2 Operational Semantics

The computation rules for the operational semantics are stated in Figure 4, and the congruence rules in Figure 6. The syntax and semantics of the rules are according to standard small-step operational-semantics with premises above the line and the conclusion below. Each rule contains triples, where each triple consist of three elements, separated by bars '|', where the first element is the term, the second the variable store σ, and the last one the flow store, ϕ.

To avoid misinterpretation of the semantics, some notations need clarification. Capture-avoiding substitution is expressed using syntax $[x \mapsto t_1]t_2$, meaning the term obtained by replacing all free occurrences of identifier x in t_2 by t_1. Similar syntax is also used for store updates, where the notation $[f \mapsto n]\phi$ means the resulting flow store that maps f to n together with all other mappings from location to flow node in ϕ. Flow stores are extended using the notation $(\phi, l \mapsto n)$, meaning the flow store ϕ extended with the mapping from l to n, where $l \notin dom(\phi)$. Updates in variable stores are expressed with the corresponding notation, i.e., $(\sigma, l \mapsto v)$. Moreover, in the usual way, rules with more specific terms in patterns are selected first, e.g., for a term $t_1 t_2$, where $t_1 \in \text{Value} \subseteq \text{Term}$, rule (E-APP2) in Figure 6 is selected in favor of (E-APP1).

The most interesting rules which differ from standard untyped lambda-calculus are the last four rules in Figure 4. An example of the application of these rules is given in Figure 5. At the first step, a `flow()` term is evaluated using rule (E-FLOW-CON). This rule creates an unused flow location in the flow store ($f \notin dom(\phi)$), maps the location to a new black node, extends the flow store ϕ with this mapping, and returns the new flow store location. The left element in the new black node is a zero value of type real. Figure 5 shows

$$(\lambda x.t)v \mid \sigma \mid \phi \longrightarrow [x \mapsto v]t \mid \sigma \mid \phi \qquad \text{(E-APPABS)}$$

$$\frac{l \notin dom(\sigma)}{\texttt{var(}v\texttt{)} \mid \sigma \mid \phi \longrightarrow l \mid (\sigma, l \mapsto v) \mid \phi} \qquad \text{(E-VAR-CON)}$$

$$\frac{f \notin dom(\phi)}{\texttt{flow()} \mid \sigma \mid \phi \longrightarrow f \mid \sigma \mid (\phi, f \mapsto \mathcal{N}_{\mathcal{B}}(0, \mathcal{N}_{\mathcal{E}}))} \qquad \text{(E-FLOW-CON)}$$

$$\frac{\begin{array}{c}\mathcal{N}_{\mathcal{B}}(t_1, n_2) = \phi(f) \\ l' \notin dom(\sigma) \qquad f' \notin dom(\phi) \\ \phi' = ([f \mapsto \mathcal{N}_{\mathcal{B}}(t_1, \mathcal{N}_{\mathcal{W}}(l', n_2))]\phi, (f' \mapsto \mathcal{N}_{\mathcal{W}}(l', \mathcal{N}_{\mathcal{E}})))\end{array}}{\texttt{fork(}f\texttt{)} \mid \sigma \mid \phi \longrightarrow f' \mid (\sigma, l' \mapsto \epsilon) \mid \phi'} \qquad \text{(E-FORK-BLACK)}$$

$$\frac{\begin{array}{c}\mathcal{N}_{\mathcal{W}}(t_1, \_) = \phi(f) \\ l' \notin dom(\sigma) \qquad f' \notin dom(\phi) \\ \phi' = ([f \mapsto \mathcal{N}_{\mathcal{B}}(t_1, \mathcal{N}_{\mathcal{W}}(l', \mathcal{N}_{\mathcal{E}}))]\phi, (f' \mapsto \mathcal{N}_{\mathcal{W}}(l', \mathcal{N}_{\mathcal{E}})))\end{array}}{\texttt{fork(}f\texttt{)} \mid \sigma \mid \phi \longrightarrow f \mid (\sigma, l' \mapsto \epsilon) \mid \phi'} \qquad \text{(E-FORK-WHITE)}$$

$$\frac{v \notin \text{FLoc}}{\texttt{fork(}v\texttt{)} \mid \sigma \mid \phi \longrightarrow v \mid \sigma \mid \phi} \qquad \text{(E-FORK-RM)}$$

Figure 4: Computation rules of the operational semantics for the $\tilde{\lambda}$–calculus.

the graph representation of the flow tree. The dashed arrow states that the input flow is zero to the black node. This node corresponds to a sum to zero equation, which has not yet any outgoing flow variables (represented by edges). In the third column in Figure 5, the current state of the flow-store is shown after evaluation of the term in column one. The fact that the black node does not have any outgoing edges is shown with the empty node $\mathcal{N}_\mathcal{E}$ in the second element.

At the second step in the example, node f_0 is forked using rule (E-FORK-BLACK). This rule is chosen in favor of (E-FORK-WHITE), since $\phi(f_0)$ represents in this case a black node. The second and third premise in this rule create both a new variable location (a flow variable) and a new flow store location. As illustrated in the graph representation, a new white node is created. In the forth premise, a new ϕ' is bound, representing the store where location f_0 is updated and a new mapping from f_1 to the new white node is added.

In the third step, f_0 is forked again. In this case another white node is created and an edge is assigned between the black node and the new white node.

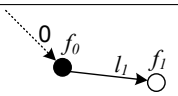
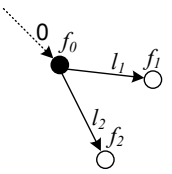
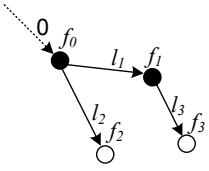
Term	**Graph Representation**	**Flow Store**	**Var Store**
`flow()` $\longrightarrow f_0$		$(f_0 \mapsto \mathcal{N}_\mathcal{B}(0, \mathcal{N}_\mathcal{E}))$	
`fork(`f_0`)` $\longrightarrow f_1$		$(f_0 \mapsto \mathcal{N}_\mathcal{B}(0, \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E})))$ $(f_1 \mapsto \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E}))$	$(l_1 \mapsto \epsilon)$
`fork(`f_0`)` $\longrightarrow f_2$		$(f_0 \mapsto \mathcal{N}_\mathcal{B}(0, \mathcal{N}_\mathcal{W}(l_2, \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E}))))$ $(f_1 \mapsto \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E}))$ $(f_2 \mapsto \mathcal{N}_\mathcal{W}(l_2, \mathcal{N}_\mathcal{E}))$	$(l_1 \mapsto \epsilon)$ $(l_2 \mapsto \epsilon)$
`fork(`f_1`)` $\longrightarrow f_3$		$(f_0 \mapsto \mathcal{N}_\mathcal{B}(0, \mathcal{N}_\mathcal{W}(l_2, \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E}))))$ $(f_1 \mapsto \mathcal{N}_\mathcal{B}(l_1, \mathcal{N}_\mathcal{W}(l_3, \mathcal{N}_\mathcal{E})))$ $(f_2 \mapsto \mathcal{N}_\mathcal{W}(l_2, \mathcal{N}_\mathcal{E}))$ $(f_3 \mapsto \mathcal{N}_\mathcal{W}(l_3, \mathcal{N}_\mathcal{E}))$	$(l_1 \mapsto \epsilon)$ $(l_2 \mapsto \epsilon)$ $(l_3 \mapsto \epsilon)$

Figure 5: Example of the fork command and respresentations in the flow store and the variable store.

Finally, step four forks the node located by f_1. Since this node is a white-node (before evaluation of $\texttt{fork}(f_1)$), rule (E-FORK-WHITE) applies. The main difference in this rule compared to (E-FORK-BLACK) is that the color of the node pointed to by f_1 is changed from white to black. This means that this fork operation both generated a new sum to zero equation (the black node) and added a flow variable l_3.

After evaluation of the given example, the flow store contain four mappings, where two of them maps to black nodes, and two maps to white ones. All locations pointing to a black node will generate a sum to zero equation. The equation is generated by letting the left side of the equation be the first element of the black node, e.g., in node $\mathcal{N}_{\mathcal{B}}(0, \mathcal{N}_{\mathcal{W}}(l_1, \mathcal{N}_{\mathcal{E}}))$, zero will be on the left hand side of the equation. The right hand side consist of the sum of white nodes term values given in element two of the black node. In the example given in Figure 5, the sum to zero equations for the final value of the flow store would be:

$$0 = l_2 + l_1 \tag{1}$$
$$l_1 = l_3 \tag{2}$$

Implicit dereferencing of locations is assumed in the above equations. These equations together with resulting equations after evaluation forms the final equation system. The variables in the equation system correspond to all locations

$$\frac{t_1 \mid \sigma \mid \phi \longrightarrow t_1' \mid \sigma' \mid \phi'}{t_1 t_2 \mid \sigma \mid \phi \longrightarrow t_1' t_2 \mid \sigma' \mid \phi'} \quad \text{(E-APP1)}$$

$$\frac{t_2 \mid \sigma \mid \phi \longrightarrow t_2' \mid \sigma' \mid \phi'}{v_1 t_2 \mid \sigma \mid \phi \longrightarrow v_1 t_2' \mid \sigma' \mid \phi'} \quad \text{(E-APP2)}$$

$$\frac{t \mid \sigma \mid \phi \longrightarrow t' \mid \sigma' \mid \phi'}{\texttt{var(}t\texttt{)} \mid \sigma \mid \phi \longrightarrow \texttt{var(}t'\texttt{)} \mid \sigma' \mid \phi'} \quad \text{(E-VAR)}$$

$$\frac{t_1 \mid \sigma \mid \phi \longrightarrow t_1' \mid \sigma' \mid \phi'}{t_1 \texttt{=} t_2 \mid \sigma \mid \phi \longrightarrow t_1' \texttt{=} t_2 \mid \sigma' \mid \phi'} \quad \text{(E-EQ1)}$$

$$\frac{t_2 \mid \sigma \mid \phi \longrightarrow t_2' \mid \sigma' \mid \phi'}{v_1 \texttt{=} t_2 \mid \sigma \mid \phi \longrightarrow v_1 \texttt{=} t_2' \mid \sigma' \mid \phi'} \quad \text{(E-EQ2)}$$

$$\frac{t_1 \mid \sigma \mid \phi \longrightarrow t_1' \mid \sigma' \mid \phi'}{t_1 \wedge t_2 \mid \sigma \mid \phi \longrightarrow t_1' \wedge t_2 \mid \sigma' \mid \phi'} \quad \text{(E-CONJ1)}$$

$$\frac{t_2 \mid \sigma \mid \phi \longrightarrow t_2' \mid \sigma' \mid \phi'}{v_1 \wedge t_2 \mid \sigma \mid \phi \longrightarrow v_1 \wedge t_2' \mid \sigma' \mid \phi'} \quad \text{(E-CONJ2)}$$

$$\frac{t \mid \sigma \mid \phi \longrightarrow t' \mid \sigma' \mid \phi'}{\texttt{fork(}t\texttt{)} \mid \sigma \mid \phi \longrightarrow \texttt{fork(}t'\texttt{)} \mid \sigma' \mid \phi'} \quad \text{(E-FORK)}$$

Figure 6: Congruence rules of the operational semantics for the $\tilde{\lambda}$–calculus.

bop	$\in \text{Bop} = \{+, -, *, /\}$	Binary operations
p	$\in$ Pattern	**Pattern**
	$p ::=$	
	x	Identifier pattern
	$\mid$ `(`$p_i{}^{i \in 1..n}$`)`	Tuple pattern
t	$\in$ Term	**Terms**
	$t ::=$	
	`(`$t_i{}^{i \in 1..n}$`)`	Tuple
	$\mid$ `func` p `{`t`}`	Function abstraction with pattern
	$\mid$ `model` p `{`t`}`	Model abstraction with pattern
	$\mid$ t_1 bop t_2	Binary operations
	$\mid$ `-`t	Uniary negation
	$\mid$ `der(`t`)`	Derivative
	$\mid$ `sin(`t`)`	Sine
	$\mid$ `cos(`t`)`	Cosine
	$\mid$ `time`	Global simulation time
v	$\in$ Value	**Values**
v	$::=$	
	`(`$v_i{}^{i \in 1..n}$`)` $\mid$ `func` p `{`t`}`	
	$\mid$ `-`v $\mid$ `der(`v`)` $\mid$ `sin(`v`)`	
	$\mid$ `cos(`v`)` $\mid$ v_1 bop v_2 $\mid$ `time`	

Figure 7: Abstract syntax of the kernel language MKL, which represents extensions to the syntax given in Figure 3.

created in the variable store. Note that this store now contains both potential variables created using the term `var(`t`)` and flow variables generated due to forking both black and white nodes.

The congruence rules in Figure 6 are less interesting, but equally important to the semantics. We have chosen to write out all the rules explicitly for completeness, even if there exist simpler and more compact ways of describing these kinds of rules. It should be noted that the congruence rules for equations (E-EQ1) and (E-EQ2), and the rules for conjunction (E-CONJ1) and (E-CONJ2) are stated with two terms to show the evaluation order.

We choose to describe the semantics with small-step-semantics, since it has been shown to exist efficient ways of proving type safety of a language using the progress and preservation theorems [14], if the language is extended with a static type system.

4 Modeling Kernel Language

To enable realistic modeling capabilities, the $\tilde{\lambda}$-calculus needs to be extended with more convenient constructs for modeling. The language presented in this section, called modeling kernel language (MKL) is then used for demonstrating modeling capabilities in Section 2.

New evaluation rules:

$$(\texttt{func}\ p\ \{t\})v \mid \sigma \mid \phi \longrightarrow match(p, v)t \mid \sigma \mid \phi \quad \text{(E-APPABS-MATCH)}$$

$$(\texttt{model}\ p\ \{t\})v \mid \sigma \mid \phi \longrightarrow (\texttt{func}\ p\ \{t\})(\texttt{fork}(v)) \mid \sigma \mid \phi \quad \text{(E-APPMODEL)}$$

$$\texttt{fork}((t^{i\in 1..n})) \mid \sigma \mid \phi \longrightarrow (\texttt{fork}(t_i)^{i\in 1..n}) \mid \sigma \mid \phi$$

$$\frac{t_j \mid \sigma \mid \phi \longrightarrow t'_j \mid \sigma' \mid \phi' \qquad t_{tmp} = (v_i{}^{i\in 1..j-1}, t'_j, t_k{}^{k\in j+1..n})}{(v_i{}^{i\in 1..j-1}, t_j, t_k{}^{k\in j+1..n}) \mid \sigma \mid \phi \longrightarrow t_{tmp} \mid \sigma' \mid \phi'} \quad \text{(E-TUPLE)}$$

Matching rules:

$$match(x, v) = [x \mapsto v] \quad \text{(M-IDENT)}$$

$$\frac{\text{for each } i \qquad match(p_i, v_i) = \rho_i}{match((p_i{}^{i\in 1..n}), (v_i{}^{i\in 1..n})) = \rho_1 \circ \cdots \circ \rho_n} \quad \text{(M-TUPLE)}$$

Figure 8: Additional semantic rules for the kernel language.

4.1 Abstract Syntax

The extra terms and syntactic categories for constructs of the extended language, are listed in Figure 7.

Several of the introduced terms are used for making the language more expressive. For example, a new syntactic category of *patterns* is introduced. The current minimal language supports identifier and tuple patterns, but the language could easily be enriched with other constructs such as records and variants.

Another term for functional abstraction, `func` p `{`t`}` has been added to distinguish it from the lambda abstraction given in the λ-calculus. The main difference is that `func` p `{`t`}` includes a pattern as its formal parameter, while the lambda absraction $\lambda x.t$ used an identifier as formal parameter.

The most important term in the MKL is the `model`-term. The purpose with this term is to create an abstraction mechanism for equation-systems using a functional modeling style, and at the same time hide the existence of the fork semantics, which is needed for correct flow semantics.

The other terms, e.g., binary operations, time derivative operation, Sine function etc., are needed to be able to create relevant models. Some of these terms could also have been implemented as library functions (e.g., Sine and Cosine), but are here part of the language for presentation purpose.

4.2 Operational Semantics

The new evaluation rules for MKL are given in Figure 8. Besides these semantic rules, some syntactic sugar is also added. For example, the `def` construct is transformed into a combination of lambda abstraction and application terms.

We will not discuss this syntactic transformation any further, since it is not important in regards to the flow connection semantics.

The functional application rule (E-APPABS-MATCH) states ordinary function application, but with pattern matching. The matching rules are expressed with a separate set of inference rules, where (M-IDENT) is used for identifier patterns and (M-TUPLE) for tuple patterns.

The most important rule of the new rules is (E-APPMODEL), which matches an application, where the first term is a model. From the definition of `model` p `{`t`}`, we can see that it is almost the same as a functional abstraction, but if we take a closer look at rule (E-APPMODEL), we note that the model is transformed into a function abstraction (a lambda abstraction with pattern), together with a `fork(`v`)` term on the second part of the application term. This construct is the key element of hiding the `fork` construct from the user. The intuition is that each time a connection should be stated between model instances, the wires (connections) need to be forked to form correct flow trees.

Finally, there is one rule (E-FORKTUPLE), which propagates the fork term into a tuple's elements, and a new congruence rule for evaluating a tuple's elements.

5 Prototype Implementation and Evaluation

To evaluate the described language semantics, a prototype implementation was constructed, where the semantic rules were directly translation into OCaml source code. The implementation is not intended for performance evaluation, but to verify the correctness of the given rules.

There are certain properties of the given semantics that we want to prove correct, but this is left to future research. However, it is not obvious how we can prove that it actually models certain properties physically correct in a domain. One alternative would be to prove properties relating to e.g., Modelica and the $\tilde{\lambda}$-calculus. However, since there does not exist any formal semantics of Modelica, which is small enough to reason about, we see this as a difficult strategy to follow.

Instead, the prototype implementation is used for verifying that relevant physical models can indeed be simulated and that they generate approximately the same simulation result. In this prototype implementation, the elaboration procedure transforms a model definition (e.g., the circuit in Section 2) to a flat set of equations. This latter representation can be converted to a flat Modelica file, which we are using for simulating the system. A number of test models were created in both Modelica and in MKL and the simulation result was compared. The purposes of these verification tests are:

- To verify that the prototype can generate equation systems that are solvable.
- To verify that the simulation result correspond to the simulation of equivalent Modelica model.

Tests have been performed on a number of models with positive result. However, it should be noted that the correctness of the current semantics is not verified comprehensively enough. Furthermore, certain proves of correctness must also be conducted in future work.

6 Related Work

The most closely related work to our flow connection semantics is the connection semantics described in the specification of the Modelica language [9]. In Modelica, connections between components (model instances) are declared by using `connect`-equations. For example, consider the following Modelica source code, which expresses the same model `Circuit`, as described in Section 2.

```
model Circuit
  Resistor  R1(R=10);
  Capacitor C(C=0.01);
  Resistor  R2(R=100);
  Inductor  L(L=0.1);
  VsourceAC AC;
  Ground    G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n,  AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n,  C.n);
  connect(AC.n, G.p);
end Circuit;
```

From a modeling perspective, connections between components are in Modelica expressed by stating one `connect`-equation between each connector (port). On the contrary, in MKL, a wire is declared, which is then connected by using the name of the wire to express the connection. From a modeling point of view, different users may have different preferences and options on what is simpler and more clear than the other. There are differences regarding modeling capabilities, but it need further analysis to conclude anything about clarity and expressiveness. However, we believe that the $\tilde{\lambda}$-calculus semantics is cleaner due to its declarative nature, which enables better ability to reason about the semantics.

Currently, it does not exist any clean small formal semantics of the Modelica language. There exist specification attempts to specify the whole language using natural semantics [6, 7]. However, this resulted in a very large formal specification, which was very hard to reason about.

Other hybrid languages, such as χ has formal operational semantics defined [13]. However, until this date, the χ language do not yet support the concept of flow connections.

A similar idea of using functional abstraction for modeling of acausal physical models were outlined by Nilsson et. al. [10]. This paradigm, which they call *functional hybrid modeling (FHM)* introduces the concept of *first-class relations on signals and switch constructs*. The signal relations `sigrel` used in the examples in the article have similarities with our model notation, but since the work by Nilsson et.al [10] does not contain any formal semantics, it is hard to analyze the exact similarities. One major difference is that Nilssons et. al.'s work does not incorporate the flow connection semantics into the semantic framework.

To the best of our knowledge, there are no previous published work of a formal semantics of encoding the flow connection semantics in the lambda calculus.

7 Conclusions

We have in this paper described a novel approach of encoding the physical flow connection semantics into the untyped lambda-calculus, using small-step operational semantics. A minimal calculus, called *flow lambda calculus*, denoted $\tilde{\lambda}$-calculus was defined. Based on this calculus, the syntax and semantics was extended to give better modeling capabilities. This language, called modeling kernel language (MKL), was demonstrated with a couple of examples. A prototype implementation of the language was implemented as an interpreter, and some models were simulated and compared with models created in the Modelica language.

Acknowledgments

I would like to thank Peter Fritzson and Björn Lisper for many helpful and constructive comments on drafts of this paper.

This research work was funded by CUGS (the National Graduate School in Computer Science, Sweden), by SSF under the VISIMOD II project, and by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project.

References

[1] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, December 1994.

[2] Paul Inigo Barton. *The Modelling and Simulation of Combined Discrete/-Continuous Processes*. PhD thesis, Department of Chemical Engineering, Imperial Collage of Science, Technology and Medicine, London, UK, 1992.

[3] Ernst Christen and Kenneth Bakalar. VHDL-AMS - A Hardware Description Language for Analog and Mixed-Signal Applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(10):1263–1272, 1999.

[4] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[5] Georgina Fábián. *A Language and Simulator for Hybrid Systems*. PhD thesis, Institute for Programming research and Algorithmics, Technische Universiteit Eindhoven, Netherlands, Netherlands, 1999.

[6] David Kågedal. A Natural Semantics specification for the equation-based modeling languge Modelica. Master's thesis, Linköping University, 1998.

[7] David Kågedal and Peter Fritzson. Generating a Modelica Compiler from Natural Semantics Specifications. In *Proceedings of the Summer Computer Simulation Conference*, 1998.

[8] MathWorks. The Mathworks - Simulink - Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/ [Last accessed: November 8, 2007].

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

[10] Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling. In *Practical Aspects of Declarative Languages : 5th International Symposium, PADL 2003*, volume 2562 of *LNCS*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.

[11] M. Oh and Costas C. Pantelides. A modelling and Simulation Language for Combined Lumped and Distributed Parameter Systems. *Computers and Chemical Engineering*, 20(6–7):611–633, 1996.

[12] Bjarne Stroustrup. A history of C++ 1979–1991. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 271–297, New York, USA, 1993. ACM Press.

[13] D.A. van Beek, K.L. Man, MA. Reniers, J.e. Rooda, and R.R.H Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *The Journal of Logic and Algebraic Programming*, 68:129–210, 2006.

[14] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.