Higher-Order Acausal Models

David Broman Peter Fritzson

Department of Information and Computer Science, Linköping University, Sweden
{davbr,petfr}@ida.liu.se

Abstract

Current equation-based object-oriented (EOO) languages typically contain a number of fairly complex language constructs for enabling reuse of models. However, support for model transformation is still often limited to scripting solutions provided by tool implementations. In this paper we investigate the possibility of combining the well known concept of higher-order functions, used in standard functional programming languages, with acausal models. This concept, called Higher-Order Acausal Models (HOAMs), simplifies the creation of reusable model libraries and model transformations within the modeling language itself. These transformations include general model composition and recursion operations and do not require data representation/reification of models as in metaprogramming/metamodeling. Examples within the electrical and mechanical domain are given using a small research language. However, the language concept is not limited to a particular language, and could in the future be incorporated into existing commercially available EOO-languages.

Keywords Higher-Order, Acausal, Modeling, Simulation, Model Transformation, Equations, Object-Oriented, EOO

1. Introduction

Modeling and simulation have been an important application area for several successful programming languages, e.g., Simula [6] and C++ [24]. 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 [15] 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 past two decades, a new kind of language has emerged, where differential algebraic equations (DAEs) can describe the continuous-time behavior of a system. Moreover, such languages often support hybrid DAEs for modeling combined continuous-time and discrete-time behavior.

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

Copyright is held by the author/owner(s). The proceedings are published by Linköping University Electronic Press. Proceedings available at:
`http://www.ep.liu.se/ecp/029/`

EOOLT 2008 website:
`http://www.eoolt.org/2008/`

These languages enable modeling of complex physical systems by combining different domains, such as electrical, mechanical, and hydraulic. Examples of such languages are Modelica [10, 17], Omola [1], gPROMS [3, 20], VHDL-AMS [5], and χ (Chi) [13, 27].

A fundamental construct in most of these languages is the *acausal model*. Such a model can encapsulate and compose both continuous-time behavior in form of DAEs and / or other interconnected sub-models, where the direction of information flow between the sub-models is not specified. Several of these languages (e.g., Modelica and Omola) support object-oriented concepts that enable the composition and reuse of acausal models. However, the possibilities to perform *transformations* on models and to create generic and reusable transformation libraries are still usually limited to tool-dependent scripting approaches [7, 11, 26], despite recent development of metamodeling/metaprogramming approaches like MetaModelica [12].

In functional programming languages, such as Haskell [23] and Standard ML [16], standard libraries have for a long time been highly reusable, due to the basic property of having functions as first-class values. This property, also called *higher-order functions*, means that functions can be passed around in the language as any other value.

In this paper, we investigate the combination of acausal models with higher-order functions. We call this concept *Higher-Order Acausal Models (HOAMs)*.

A similar idea called *first-class relations on signals* has been outlined in the context of functional hybrid modeling (FHM)[18]. However, the work is still at an early stage and it does not yet exist any published description of the semantics. By contrast, our previous work's main objective has been to define a formal operational semantics for a subset of a typical EOO language [4]. From the technical results of our earlier work, we have extracted the more general ideas of HOAM, which are presented in this paper in a more informal setting.

An objective of this paper is to be accessible both to engineers with little functional language programming background, as well as to computer scientists with minimal knowledge of physical acausal modeling. Hence, the paper is structured in the following way to reflect both the broad intended audience, as well as presenting the contribution of the concept of HOAMs:

- The fundamental ideas of traditional higher-order functions are explained using simple examples. Moreover, we give the basic concepts of acausal models when used for modeling and simulation (Section 2).
- We state a definition of higher order acausal models (HOAMs) and outline motivating examples. Surprisingly, this concept has not been widely explored in the context of EOO-languages (Section 2).
- The paper gives an informal introduction to physical modeling in our small research language called Modeling Kernel Language (MKL) (Section 3).
- We give several concrete examples within the electrical and mechanical domain, showing how HOAMs can be used to create highly reusable modeling and model transformation/composition libraries (Section 4).

Finally, we discuss future perspectives of higher-order acausal modeling (Section 5), and related work (Section 6).

2. The Basic Idea of Higher-Order

In the following section we first introduce the well established concept of anonymous functions and the main ideas of traditional higher-order functions. In the last part of the section we introduce acausal models and the idea of treating models with acausal connections to be higher-order.

2.1 Anonymous Functions

In functional languages, such as Haskell [23] and Standard ML [16], the most fundamental language construct is functions. Functions correspond to partial mathematical functions, i.e., a function $f : A \to B$ gives a mapping from (a subset of) the domain A to the codomain B.

In this paper we describe the concepts of higher-order functions and models using a tiny untyped research language called *Modeling Kernel Language (MKL)*. The language has similar modeling capabilities as parts of the Modelica language, but is primarily aimed at investigating novel language concepts, rather than being a full-fledged modeling and simulation language. In this paper an informal example-based presentation is given. However, a formal operational semantics of the dynamic elaboration semantics for this language is available in [4].

In MKL, similar to general purpose functional languages, functions can be defined to be *anonymous*, i.e., the function is defined without an explicit naming. For example, the expression

```
func(x){x*x}
```

is an anonymous function that has a formal parameter `x` as input parameter and returns `x` squared[1]. Formal parameters are written within parentheses after the `func` keyword, and the expression representing the body of the function is given within curly parentheses; in this case `{x*x}`.

An anonymous function can be applied by writing the function before the argument(s) in a parenthesized list, e.g. `(3)`:

```
func(x){x*x}(3)
→ 3*3
→ 9
```

The lines starting with a left arrow (→) show the evaluation steps when the expression is executed.

However, it is often convenient to name values. Since anonymous functions are treated as values, they can be defined to have a name using the `def` construct in the same way as constants.

```
def pi = 3.14
def power2 = func(x){x*x}
```

Here, both `pi` and function `power2` can be used within the defined scope. Hence, the definitions can be used to create new expressions for evaluation, for example:

```
power2(pi)
→ power2(3.14)
→ 3.14 * 3.14
→ 9.8596
```

2.2 Higher-Order Functions

In many situations, it is useful to pass a function as an argument to another function, or to return a function as a result of executing a function. When functions are treated as values and can be passed around freely as any other value, they are said to be *first-class citizens*. In such a case, the language supports *higher-order functions*.

DEFINITION 1 (Higher-Order Function).
A higher-order function is a function that

1. *takes another function as argument, and/or*
2. *returns a function as the result*

Let us first show the former case where functions are passed as values. Consider the following function definition of `twice`, which applies the function `f` two times on `y`, and then returns the result.

```
def twice = func(f,y){
    f(f(y))
};
```

The function `twice` can then be used with an arbitrary function `f`, assuming that types match. For example, using it in combination with `power2`, this function is applied twice.

```
twice(power2,3)
→ power2(power2(3))
→ power2(3*3)
→ power2(9)
→ 9*9
→ 81
```

Since `twice` can take any function as an argument, we can apply `twice` to an anonymous function, passed directly as an argument to the function `twice`.

[1] In programming language theory, an anonymous function is called a *lambda abstraction*, written $\lambda x.e$, where x is the formal parameter and e is the expression representing the body of the function. The corresponding syntactic form in MKL for a lambda abstraction is `func` $p\{e\}$, where *p* is a *pattern*. A pattern can be a *n*-ary tuple enclosed in parenthesis, e.g., a tuple pattern with one parameter can have the form `(x)` and one with two parameters `(x,y)`.

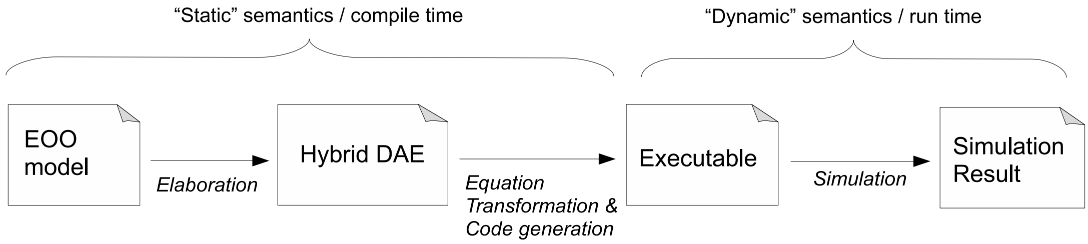


Figure 1. Outline of a typical compilation and simulation process for an EOO language tool.

```
twice(func(x){2*x-3},5)
→ func(x){2*x-3}(func(x){2*x-3}(5))
→ func(x){2*x-3}(2*5-3)
→ func(x){2*x-3}(7)
→ 2*7-3
→ 11
```

Let us now consider the second part of Definition 1, i.e., a function that returns another function as the result.

In mathematics, functional composition is normally expressed using the infix operator $\circ$. Two functions $f : X \to Y$ and $g : Y \to Z$ can be composed to $g \circ f : X \to Z$, by using the definition $(g \circ f)(x) = g(f(x))$.

The very same definition can be expressed in a language supporting higher-order functions:

```
def compose = func(g,f){
    func(x){g(f(x))}
};
```

This example illustrates the creation of a new anonymous function and returning it from the `compose` function. The function composes the two functions given as parameters to `compose`. Hence, this example illustrates both that higher-order functions can be applied to functions passed as arguments (using formal parameters `f` and `g`), and that new functions can be created and returned as results (the anonymous function).

To illustrate an evaluation trace of the composition function, we first define another function `add7`

```
def add7 = func(x){7+x};
```

and then compose `power2` and `add7` together, forming a new function `foo`:

```
def foo = compose(power2,add7);
→ def foo = func(x){power2(add7(x))};
```

Note how the function `compose` applied to `power2` and `add7` evaluates to an anonymous function. Now, the new function `foo` can be applied to some argument, e.g.,

```
foo(4)
→ func(x){power2(add7(x))}(4)
→ power2(add7(4))
→ power2(7+4)
→ power2(11)
→ 11*11
→ 121
```

The simple numerical examples given here only show the very basic principle of higher-order functions. In functional programming other more advanced usages, such as list manipulation using functions `map` and `fold`, are very common.

2.3 Elaboration and Simulation of Acausal Models

In conventional object-oriented programming languages, such as Java or C++, the behavior of classes is described using methods. On the contrary, in equation-based object-oriented languages, the continuous-time behavior is typically described using differential algebraic equations and the discrete-time behavior using constructs generating events. This behavior is grouped into abstractions called classes or models (Modelica) or entities and architectures (VHDL-AMS). From now on we refer to such an abstraction simply as *models*.

Models are blue-prints for creating *model instances* (in Modelica called components). The models typically have well-defined interfaces consisting of ports (also called connectors), which can be connected together using *connections*. A typical property of EOO-languages is that these connections usually are *acausal*, meaning that the direction of information flow between model instances is not defined at modeling time.

In the context of EOO languages, we define acausal (also called non-causal) models as follows:

DEFINITION 2 (Acausal Model).
An acausal model is an abstraction that encapsulates and composes

1. *continuous-time behavior in form of differential algebraic equations (DAEs)*
2. *other interconnected acausal models, where the direction of information flow between sub-models is not specified.*

In many EOO languages, acausal models also contain conditional constructs for handling discrete events. Moreover, connections between model instances can typically both express potential connections (across) and flow (also called through) connections generating sum-to-zero equations. Examples of acausal models in both MKL and Modelica are given in Figure 2 and described in Section 3.1.

A typical implementation of an EOO language, when used for modeling and simulation, is outlined in Figure 1. In the first phase, a hierarchically composed acausal model is *elaborated* (also called flattened or instantiated) into a hybrid DAE, describing both continuous-time behavior (DAEs) and discrete-time behavior (e.g., when-equations). The second phase performs *equation transformations and* *code generation*, which produces executable target code. When this code is executed, the actual simulation of the model takes place, which produces a simulation result. In the most common implementations, e.g., Dymola [7] or OpenModelica [26], the first two phases occur during compile time and the simulation can be viewed as the run-time. However, this is not a necessary requirement of EOO languages in general, especially not if the language supports structurally dynamic systems (e.g., Sol [29], FHM [18], or MOSILAB [8]).

2.4 Higher-Order Acausal Models

In EOO languages models are typically treated as compile time entities, which are translated into hybrid DAEs during the elaboration phase. We have previously seen how functions can be turned into first-class citizens, passed around, and dynamically created during evaluation. Can the same concept of higher-order semantics be generalized to also apply to acausal models in EOO languages? If so, does this give any improved expressive power in such generalized EOO language?

In the next section we describe concrete examples of acausal modeling using MKL. However, let us first define what we actually mean by higher-order acausal models.

DEFINITION 3 (Higher-Order Acausal Model (HOAM)). *A higher-order acausal model is an acausal model, which can be*

1. *parametrized with other HOAMs.*
2. *recursively composed to generate new HOAMs.*
3. *passed as argument to, or returned as result from functions.*

In the first case of the definition, models can be parametrized by other models. For example, the constructor of a automobile model can take as argument another model representing a gearbox. Hence, different automobile instances can be created with different gearboxes, as long as the gearboxes respects the interface (i.e., type) of the gearbox parameter of the automobile model. Moreover, an automobile model does not necessarily need to be instantiated with a specific gearbox, but only *specialized* with a specific gearbox model, thus generating a new more specific model.

The second case of Definition 3 states that a model can reference itself; resulting in a recursive model definition. This capability can for example express models composed of many similar parts, e.g., discretization of flexible shafts in mechanical systems or pipes in fluid models.

Finally, the third case emphasizes the fact that HOAMs are first-class citizens, e.g., that models can be both passed as arguments to functions and created and returned as results from functions. Hence, in the same way as in the case of higher-order functions, generic reusable functions can be created that perform various tasks on arbitrary models, as long as they respect the defined types (interfaces) of the models' formal parameters. Consequently, this property enables *model transformations* to be defined and executed within the modeling language itself. For example, certain discretizations of models can be implemented as a generic function and stored in a standard library, and then reused with different user defined models.

Some special and complex language constructs in currently available EOO languages express part of the described functionality (e.g., the redeclare and for-equation constructs in Modelica). However, in the next sections we show that the concept of acausal higher-order models is a small, but very powerful and expressive language construct that subsumes and/or can be used to define several other more complex language constructs. If the end user finds this more functional approach of modeling easy or hard depends of course on many factors, e.g., previous programming language experiences, syntax preferences, and mathematical skills. However, from a semantic point of view, we show that the approach is very expressive, since few language constructs enable rich modeling capabilities in a relatively small kernel language.

3. Basic Physical Modeling in MKL

To concretely demonstrate the power of HOAMs, we use our tiny research language Modeling Kernel Language (MKL). The higher-order function concept of the language was briefly introduced in the previous section. In this section we informally outline the basic idea of physical modeling in MKL; a prerequisite for Section 4, which introduces higher-order acausal models using MKL.

3.1 A Simple Electrical Circuit

To illustrate the basic modeling capabilities of MKL, the classic simple electrical circuit model is given in Figure 2. Part (I) shows the graphical layout of the model and (II) shows the corresponding textual model given in MKL. For clarity to the readers familiar with the Modelica language, we also compare with the same model given as Modelica textual code (III).

In MKL, models are always defined anonymously. In the same way as for anonymous functions, an anonymous model can also be given a name, which is in this example done by giving the model the name `circuit`. The model takes zero formal parameters, given by the empty tuple (parenthesized list) to the right of the keyword `model`. The contents of the model is given within curly braces. The first four statements define four new *wires*, i.e., connection points from which the different components (model instances) can be connected.

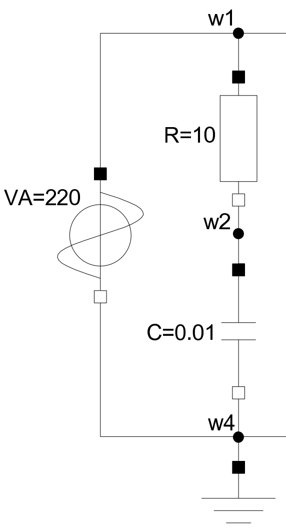
The six components defined in this circuit correspond to the layout given in part (I) in Figure 2. Consider the first resistor instantiated using the following:

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

The two first arguments state that wires `w1` and `w2` are connected to this resistor. The last argument expresses that the resistance for this instance is 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`.

Modeling using MKL differs in several ways compared to Modelica (Figure 2, part III). First, models are not defined anonymously in Modelica and are not treated as first-class citizens. Second, the way acausal connections are de-

(I)



(II)

```
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);
};
```

(III)

```
model Circuit
    Resistor  R1(R=10);
    Capacitor C(C=0.01);
    Resistor  R2(R=100);
    Inductor  L(L=0.1);
    VsourceAC AC(VA=220);
    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;
```

Figure 2. Model of a simple electrical circuit. Figure part (I) shows the graphical model of the circuit, (II) gives the corresponding MKL model definition, and (III) shows a Modelica model of the same circuit.

fined between model instances differs. In MKL, the connection (in this electrical case a wire), is created and then connected to the model instances by giving it as arguments to the creation of sub-model instances. In Modelica, a special `connect`-equation construct is defined in the language. This construct is used to define binary connections between connectors of sub-model instances. From a user point of view, both approaches can be used to express acausal connections between model instance. Hence, we let it be up to the reader to judge what is the most natural way of defining interconnections. However, from a formal semantics point of view, in regards to HOAMs, we have found it easier to encode connections using ordinary parameter passing style[2].

3.2 Connections, Variables, and Flow Nodes

The concept of wire is not built into the language. Instead, it is defined using an anonymous function, referring to the built-in constructs `var()` and `flow()`:

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

Here, a function called `Wire` is defined by using the anonymous function construct `func`. The definition states that the function has an empty formal parameter list (i.e., takes an empty tuple `()` as argument) and returns a tuple `(var(),flow())`, consisting of two elements. A tuple is expressed as a sequence of terms separated by commas and enclosed in parentheses.

[2] In the technical report [4], we have been able to define the elaboration semantics with HOAMs using an effectful small-step operational semantics. The main challenge of handling HOAMs and acausal connections concerns the treatment of flow variables and sum-to-zero equation. By using the parameter passing style, we avoid Modelica's informal semantic approach of using connection-sets. Moreover, by using this approach, the generated sum-to-zero equations implicitly gets the right signs, without the need of keeping track of outside/inside connectors.

The first element of the defined tuple expresses the creation of a new unknown continuous-time variable using the syntax `var()`. The variable could also been assigned 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 formal semantics of [4]. However, in this informal introduction, we just accept that Kirchhoff's current law with sum to zero at nodes is managed in a correct way.

In the circuit definition (Figure 2, part II) we used the syntax `Wire()`, which means that the function is invoked without arguments. The function call returns the tuple `(var(),flow())`. Hence, the `Wire` definition is used for encapsulating the tuple, allowing the definition to be reused without the need to restate its definition over and over again.

3.3 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;
};
```

In the same way as for `Circuit`, these sub-models are defined anonymously using the keyword `model` followed by a formal parameter and the model's content stated within curly braces. A formal parameter can be a pattern and *pattern matching* [3] 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 are not discovered until 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 inductor model is defined similarly to the `Resistor` model:

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

The main difference to the `Resistor` model is that the `Inductor` model contains a differential equation `L*der(pi) = v;`, where the `pi` variable is differentiated with respect to time using the built-in `der` operator.

The other sub-models shown in this example (`Ground`, `VSourceAC`, and `Capacitor`) is defined in a similar manner as the one above.

[3] A pattern can be a variable name, an underscore, or a tuple. When argument values are passed, each value is matched against its corresponding pattern. A variable is bound to the corresponding argument value, an underscore matches anything, i.e., nothing happens; a tuple is matched against a tuple value resulting in that each variable name in the tuple pattern is bound to the corresponding value in the tuple.

3.4 Executing the Model

Recall Figure 1, which outlined the compilation and simulation process for a typical EOO language. When a model is evaluated (executed) in MKL, this means the process of elaborating a model into a DAE. Hence, the steps of equation transformation, code generation, and simulation are not part of the currently defined language semantics. This latter steps can be conducted in a similar manner as for an ordinary Modelica implementation. Alternatively, the resulting equation system can be used for other purposes, such as optimization [14]. In the next section we illustrate several examples of how HOAMs can be used. Consequently, these examples concern the use of HOAMs during the elaboration phase, and not during the simulation phase. Further discussion on future aspects of HOAMs during these latter phases is given in Section 5.

4. Examples of Higher-Order Modeling

In Definition 3 (Section 2.4) we defined the meaning of HOAMs, giving three statements on how HOAMs can be used. This section is divided into sub-sections, where we exemplify these three kinds of usage by giving examples in MKL.

4.1 Parameterization of Models with Models

A common goal of model design is to make model libraries extensible and reusable. A natural requirement is to be able to parameterize models with other models, i.e., to reuse a model by replacing some of the sub-models with other models. To illustrate the main idea of parameterized acausal models, consider the following over-simplified example of an automobile model, where we use `Connection()` with the same meaning as the previous `Wire()`:

```
def Automobile = model(Engine, Tire){
   def c1 = Connection();
   def c2 = Connection();
   Engine(c1);
   Gearbox(c1,c2);
   Tire(c2); Tire(c2); Tire(c2); Tire(c2)
};
```

In the example, the automobile is defined to have two formal parameters; an `Engine` model and a `Tire` model. To create a model instance of the automobile, the model can be applied to a specific engine, e.g., a model `EngineV6` and some type of tire, e.g. `TireTypeA`:

```
Automobile(EngineV6,TireTypeA);
```

If later on a new engine was developed, e.g., `EngineV8`, a new automobile model instance can be created by changing the arguments when the model instance is created, e.g.,

```
Automobile(EngineV8,TireTypeA);
```

Hence, new model instances can be created without the need to modify the definition of the `Automobile` model. This is analogous to a higher-order function which takes a function as a parameter.

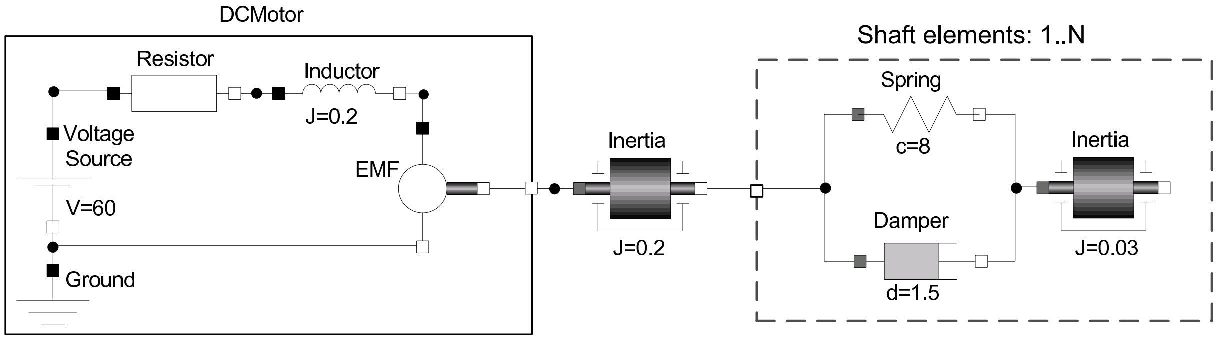


Figure 3. A mechatronic system with a direct current (DC) motor to the left and a flexible shaft to the right. The flexible shaft consists of 1 to N elements, where each element includes an inertia, a spring, and a damper.

In the example above, the definition of `Automobile` was not parametrized on the `Gearbox` model. Hence, the `Gearbox` definition must be given in the lexical scope of the `Automobile` definition. However, this model could of course also be defined as a parameter to `Automobile`.

This way of reusing acausal models has obvious strengths, and it is therefore not surprising that constructs with similar capabilities are available in some EOO languages, e.g., the special `redeclare` construct in Modelica. However, instead of creating a special language construct for this kind of reuse, we believe that HOAMs can give simpler and a more uniform semantics of a EOO language.

4.2 Recursively Defined Models

In many applications it is enough to hierarchically compose models by explicitly defining model instances within each other (e.g., the simple `Circuit` example). However, sometimes several hundreds of model instances of the same model should be connected to each other. This can of course be achieved manually by creating hundreds of explicit instances. However, this results in very large models that are hard to maintain and get an overview of.

One solution could be to add a loop-construct to the EOO language. This is the approach taken in Modelica, with the `for`-equation construct. However, such an extra language construct is actually not needed to model this behavior. Analogously to defining recursive functions, we can define *recursive models*. This gives the same modeling possibilities as adding the `for`-construct. However, it is more declarative and we have also found it easier to define a compact formal semantics of the language using this construct.

Consider Figure 3 which shows a Mechatronic model, i.e., a model containing components from both the electrical and mechanical domain. The left hand side of the model shows a simple direct current (DC) motor. The electromotoric force (EMF) component converts electrical energy to mechanical rotational energy. If we recall from Section 2, the connection between electrical components was defined using the `Wire` definition. However, in the rotational mechanical domain, the connection is instead defined by using the angle for the potential variable and the torque for flow. The rotational connection is defined as follows:

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

In the middle of the model in Figure 3 a rotational body with Inertia *J=0.2* is defined. This body is connected to a flexible shaft, i.e., a shaft which is divided into a number of small bodies connected in series with a spring and a damper in parallel in between each pair of bodies. N is the number of shaft elements that the shaft consists of.

A model of the mechatronic system is described by the following MKL source code.

```
def MechSys = model(){
    def c1 = RotCon();
    def c2 = RotCon();
    DCMotor(c1);
    Inertia(c1,c2,0.2);
    FlexibleShaft(c2,RotCon(),120);
};
```

The most interesting part is the definition of the component `FlexibleShaft`. This shaft is connected to the Inertia to the left. To the right, an empty rotational connection is created using the construction `RotCon()`, resulting in the right side not being connected. The third argument states that the shaft should consist of 120 elements.

Can these 120 elements be described without the need of code duplication? Yes, by the simple but powerful mechanism of recursively defined models. Consider the following self-explanatory definitions of `ShaftElement`:

```
def ShaftElement = model(ca,cb){
    def c1 = RotCon();
    Spring(ca,c1,8);
    Damper(ca,c1,1.5);
    Inertia(c1,cb,0.03);
};
```

This model represents just one of the 120 elements connected in series in the flexible shaft. The actual flexible shaft model is recursively defined and makes use of the `ShaftElement` model:

```
defrec FlexibleShaft = model(ca,cb,n){
    if(n==1)
        ShaftElement(ca,cb)
    else{
        def c1 = RotCon();
        ShaftElement(ca,c1);
        FlexibleShaft(c1,cb,n-1);
    };
};
```

The recursive definition is analogous to a standard recursively defined function, where the `if`-expression evaluates to false, as long as the count parameter `n` is not equal to 1. For each recursive step, a new connection is created by defining `c1`, which connects the shaft elements in series. Note that the last element of the shaft is connected to the second port of the `FlexibleShaft` model, since the shaft element created when the `if`-expression is evaluated to true takes parameter `cb` as an argument.

When the `MechSys` model is elaborated using our MKL prototype implementation, it results in a DAE consisting of 3159 equations and the same number of unknowns. It is obviously beneficial to be able to define recursive models in cases such as the one above, instead of manually creating 120 instances of a shaft element.

However, it is still a bit annoying to be forced to write the recursive model definition each time one wants to serialize a number of model instances. Is it possible to capture and define this serialization behavior once and for all, and then reuse this functionality?

4.3 Higher-Order Functions for Generic Model Transformation

In the previous section we have seen how models can be reused by applying models to other models, or to recursively define models. In this section we show that it is indeed possible to define several kinds of *model transformations* by using higher-order functions. These functions can in turn be part of a modeling language's standard library, enabling reuse of model transformation functions.

Recall the example from Section 2.2 of higher-order functions returning other anonymously defined functions. Assume that we want to create a generic function, which can take any two models that have two ports defined (`Resistor`, `Capacitor`, `ShaftElement` etc), and then compose them together by connecting them in parallel, and then return this new model:

```
def composeparallel = func(M1,M2){
    model(p,n){
        M1(p,n);
        M2(p,n);
    }
};
```

However, our model `Resistor` etc. does not take two arguments, but 3, where the last one is the value for the particular component (resistance for the `Resistor`, inductance for the `Inductor` etc.). Hence, it is convenient to define a function that sets the value of this kind of model and returns a more *specialized* model[4]:

```
def set = func(M,val){
    model(p,n){
        M(p,n,val);
    }
};
```

[4] In these examples we are using tuples as argument to the function, which makes it necessary to introduce a set function. The same kind of specialization can of course also be performed using *currying*. However, we have chosen to use the tuple notation, since it is likely to be more accessible for the reader with little experience of functional languages.

For example, a new model `Foo` that composes two other models can be defined as follows:

```
def Foo = composeparallel(set(Resistor, 100),
                          set(Inductor, 0.1));
```

A standard library can then further be enhanced with other generic functions, e.g., a function that composes two models in series:

```
def composeserial = func(M1,M2,Con){
    model(p,n){
        def w = Con();
        M1(p,w);
        M2(w,n);
    }
};
```

However, this time the function takes a third argument, namely a connector, which is used to create the connection between the models created in series. Since different domains have different kinds of connections (Wires, RotCon etc.), this must be supplied as an argument to the function. These connections are defined as higher-order functions and can therefore easily be passed as a value to the `composeserial` function.

We have now created two simple generic functions which compose models in parallel and in series. However, can we create a generic function that takes a model M, a connector C, and an integer n, and then returns a new model where n number of models M has been connected in series, using connector C? If this is possible, we do not have to create a special recursive model for the `FlexibleShaft`, as shown in the previous section.

Fortunately, this is indeed possible by combining a generic recursive model and a higher-order function. First, we define a recursive model `recmodel`:

```
defrec recmodel = model(M,C,ca,cb,n){
    if(n==1)
        M(ca,cb)
    else{
        def c1 = C();
        M(ca,c1);
        recmodel(M,C,c1,cb,n-1);
    };
};
```

Note the similarities to the recursively defined model `FlexibleShaft`. However, in this version an arbitrary model `M` is composed in series, using connector parameter `C`.

To make this model useful, we encapsulate it in a higher-order function, which takes a model `M`, a connector `C`, and an integer number `n` of the number of wanted models in series as input:

```
def serialize = func(M,C,n){
    model(ca,cb){
        recmodel(M,C,ca,cb,n);
    }
};
```

Now, we can once again define the mechatronic system given in Figure 3, but this time by using the generic function `serialize`:

```
def MekSys2 = model(){
    def c1 = RotCon();
    def c2 = RotCon();
    DCMotor(c1);
    Inertia(c1,c2,0.2);
    def FlexibleShaft =
        serialize(ShaftElement,RotCon,120);
    FlexibleShaft(c2,RotCon());
};
```

Even if the serialize function might seem a bit complicated to define, the good news is that such functions usually are created by library developers and not end-users. Fortunately, the end-user only has to call the serialize function and then use the newly created model. For example, to create a new model, where 50 resistors are composed in series is as easy as the following:

```
def Res50 =
   serialize(set(Resistor,100), Wire, 50);
```

5. Future Perspectives of Higher-Order Modeling

Our current design of higher-order acausal modeling capabilities as presented here is restricted to executing during the compiler (or interpreter) model elaboration phase, i.e., it cannot interact with run-time objects during simulation. However, removing this restriction gives interesting possibilities for run-time higher-order acausal modeling:

- The run-time results of simulation can be used in conjunction with models as first-class objects in the language, i.e., run-time creation of models, composition of models, and returning models. This is also useful in applications such as model-based optimization or model-based control, influenced by results from (on-line) simulation of models, e.g., [9].
- Structural variability [8, 18, 19, 29] of models and systems of equations means that the model structure can change at run-time, e.g., change in causality and/or number of equations. Run-time support for higher-order acausal model can be seen as a general approach to structurally variable systems. These ideas are discussed in [18, 19] in the context of Functional Hybrid Modeling (FHM).

These run-time modeling facilities provide more flexibility and expressive power but also give rise to several research challenges that need to be addressed:

- How can static strong type checking be preserved?
- How can high performance from compile-time optimizations be preserved? One example is index reduction, which requires symbolic manipulation of equations.
- How can we define a formal sound semantics for such a language?

Another future generalization of higher-order acausal modeling would be to allow models to be propagated along connections. For example, a water source could be connected to a generic flow connection structure with unspecified media. The selection of a media of type water in the source would automatically propagate to other objects.

6. Related Work

The main emphasis of this work is to explore the language concept of HOAMs in the context of EOO languages. In the following we briefly discuss three aspects of work which is related to this topic.

6.1 Functional Hybrid Modeling

As mentioned in the introduction, our notation of HOAMs has similarities to *first-class relations on signals*, as outlined in the context of Functional Hybrid Modeling (FHM) [18, 19]. The concepts in FHM are a generalization of Functional Reactive Programming (FRP) [28], which is based on reactive programming with causal hybrid modeling capabilities. Both FHM and FRP are based on *signals* that conceptually are functions over time. While FRP supports causal modeling, the aim of FHM is to support acausal modeling with structurally dynamic systems. However, the work of FHM is currently at an early stage and no published formal semantics or implementation currently exist.

HOAMs are similar to FHM's relations on signals in the sense that they are both first-class and that they can recursively reference themselves. In this paper we have showed how recursion can be used to define large structures of connected models, while in [19] ideas are outlined how it can be used for structurally dynamic systems.

One difference is that FHM's relations on signals are as it name states only relations on signals, while MKL acausal models can be parameterized on any type, e.g., other HOAMs or constants. By contrast, FHM's relation on signals can be parameterized by other relations or constants using ordinary functional abstraction, i.e., free variables inside a relation can be bound by a surrounding function abstraction. There are obvious syntactic differences, but the more specific semantic differences are currently hard to compare, since there are no public semantic specification available for any FHM language.

The work with MKL has currently focused on formalizing a kernel language for the elaboration process of typical EOO languages, such as Modelica. Hence, the formal semantics of MKL defined in [4] investigates the complications when HOAMs are combined with flow variables, generating sum-to-zero equations. How this kind of issue is handled in FHM is currently not published.

6.2 Metaprogramming and Metamodeling

The notion of higher-order models is related to, but different from metamodeling and metaprogramming. A metaprogram is a program that takes other programs/models as data and produces programs/models as data, i.e., meta-programs can manipulate *object programs* [21]. A metamodel may also have a subset of this functionality, i.e., it may specify the structure of other models represented as data, but not necessarily be executable and produce other models.

Staged metaprogramming can be achieved by quoting/unquoting operations applied in two or more stages, e.g., as in MetaML [25] and Template Haskell [22].

We have earlier developed a simple metaprogramming facility for Modelica by introducing quoting/unquoting mechanisms [2], but with limited ability to perform operations on code. A later extension [12] introduced general metaprogramming operations based on pattern-matching and transformations of abstract-syntax tree representations of models/programs similar to those found in many functional programming languages.

By contrast, the notion of higher-order models in this paper allows direct access to models in the language, e.g., passing models to models and functions, returning models, etc, without first representing (or viewing, reifying) models as data. This allows more integrated access to such facilities within the language including integration with the type system. Moreover, it often implies simpler usage and increased re-use compared to what is typically offered by metaprogramming approaches.

Metaprogramming, on the other hand, offers the possibility of greater generality on the allowed operations on models, e.g., symbolic differentiation of model equations, and the possibility of compile-time only approaches without any run-time penalty.

We should also mention the common usage of interpretive scripting languages, e.g., Python, or add-on interpretive scripting facilities using algorithmic parts of the modeling language itself such as in OpenModelica [12] and Dymola [7]. This works in practice, but is less well integrated and typically a bit ad hoc. This either requires two languages (e.g., Python and Modelica), or a separate interpretive implementation of a subset of the same language (e.g., Modelica scripting) which often give some differences in semantics, ad hoc restrictions, and inconsistent or partially missing integration with a general type system.

6.3 Modelica Redeclare and For-Equations

Modelica [17] provides a powerful facility called redeclaration, which has some capabilities of higher order models. Using redeclare, models can be passed as arguments to models (but not to functions using ordinary argument passing mechanisms e.g., at run-time), and returned from models in the context of defining a new model. For example:

```
model RefinedResistorCircuit =
  GenericResistorCircuit
  (redeclare model ResistorModel =
   TempResistor);
```

Redeclaration can also be used to adapt a model when it is inherited:

```
extends GenericResistorCircuit
  (redeclare model ResistorModel =
   TempResistor)
```

Redeclare is a compile-time facility which operates during the model elaboration phase. Moreover, using redeclare it is not possible to pass a model to a function, or to return a model from a function. Redeclaration is similar to C++ templates and Java Generics in that it allows passing types/models, but is more closely integrated in the language since it part of the class/model concept rather than being a completely separate feature. The Modelica redeclare can be seen as a special case of the more general concept of higher-order acausal models.

Modelica also provides the concept of for-equations to express repetitive equations and connection structures. Since iteration can be expressed as recursion, also for models as shown in Section 4.2, the concept of for-equations can be expressed as a special case of the more general concept of recursive models included in higher-order acausal models.

Even though EOO languages, such as Modelica, does not support HOAMs at the syntax level, HOAMs can still be very useful as a semantic concept for describing a precise formal semantics of the language. Language constructs, such as `for`-equations, can then be transformed down to a smaller kernel language. Having a small precisely defined language semantics can then make the language specification less ambiguous, enable better formal model checking possibilities, as well as providing more accurate model exchange.

7. Conclusions

We have in this paper informally presented how the concept of higher-order functions can be combined with acausal models. This concept, which we call higher-order acausal models (HOAMs), has been shown to be a fairly simple and yet powerful construct, which enables both parameterized models and recursively defined models. Moreover, by combining it with functions, we have briefly shown how it can be used to create reusable model transformation functions, which typically can be part of a model language's standard library. The examples and the implementation were given in a small research language called Modeling Kernel Language (MKL), and it was illustrated how HOAMs can be used during the elaboration phase. However, the concept is not limited to the elaboration phase, and we believe that future research in the area of HOAMs at runtime can enable both more declarative expressiveness as well as simplified semantics of EOO languages.

Acknowledgments

We would like to thank Jeremy Siek and the anonymous reviewers for many useful comments on this paper. This research work was funded by CUGS (the National Graduate School in Computer Science, Sweden) 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] Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus, and Kaj Nyström. Meta Programming and Function Overloading in OpenModelica. In *Proceedings of the 3rd International Modelica Conference*, pages 431–440, Linköping, Sweden, 2003.

[3] 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.

[4] David Broman. Flow Lambda Calculus for Declarative Physical Connection Semantics. Technical Reports in Computer and Information Science No. 1, LIU Electronic Press, 2007.

[5] 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.

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

[7] Dynasim. Dymola - Dynamic Modeling Laboratory (Dynasim AB). `http://www.dynasim.se/` [Last accessed: April 30, 2008].

[8] Christoph Nytsch-Geusen et. al. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.

[9] Rüdiger Franke, Manfred Rode, and Klaus Krüger. On-line Optimization of Drum Boiler Startup. In *Proceedings of the 3rd International Modelica Conference*, pages 287–296, Linköping, Sweden, 2003.

[10] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, New York, USA, 2004.

[11] Peter Fritzson, Peter Aronsson, Adrian Pop, Håkan Lundvall, Kaj Nyström, Levon Saldamli, David Broman, and Anders Sandholm. OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching. In *IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, Germany, 2006.

[12] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proceedings of the 4th International Modelica Conference*, pages 519–525, 2005.

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

[14] Johan Åkesson. *Languages and Tools for Optimization of Large-Scale Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, November 2007.

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

[16] Robin Milner, Mads Tofte, Robert Harper, and David MacQuee. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

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

[18] 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.

[19] Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling from an Object-Oriented Perspective. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 71–87, Berlin, Germany, 2007. Linköping University Electronic Press.

[20] 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.

[21] Tim Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *LNCS*, pages 2–44. Springer-Verlag, 2001.

[22] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, USA, 2002. ACM Press.

[23] Simon Peyton Jones. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[24] 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.

[25] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[26] The OpenModelica Project. `www.openmodelica.org` [Last accessed: May 8, 2008].

[27] 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.

[28] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252, New York, USA, 2000. ACM Press.

[29] Dirk Zimmer. Enhancing Modelica towards variable structure systems. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 61–70, Berlin, Germany, 2007. Linköping University Electronic Press.