

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.

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 $f : A \rightarrow B$ 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 squared¹. Formal parameters are written within parentheses after the `func` keyword,

¹In 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) .

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 (\rightarrow) 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`.

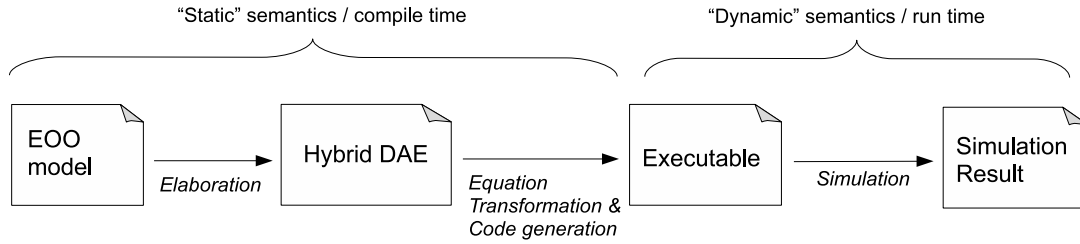


Figure 1. Outline of a typical compilation and simulation process for an EOO language tool.

```

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 $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ can be composed to $g \circ f : X \rightarrow Z$, 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-

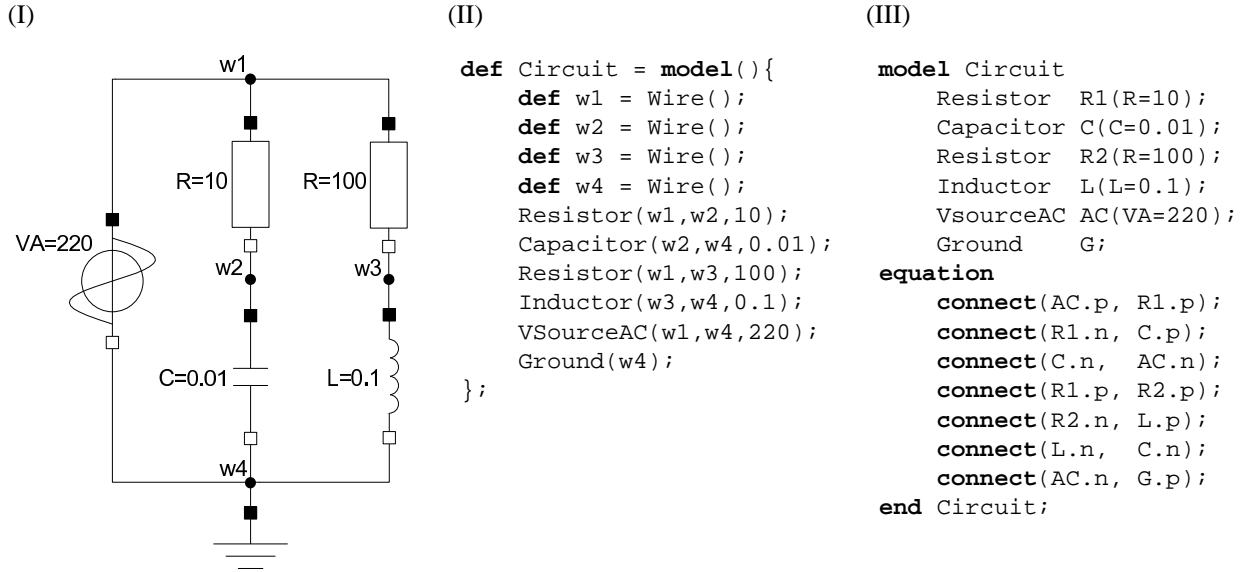


Figure 2. Model of a simple electrical circuit. Figure part (I) shows the graphical model of the circuit, (II) gives the corresponding MKL model definition, and (III) shows a Modelica model of the same circuit.

defined 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 style².

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.

²In 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 be 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;
};

```