Functional Hybrid Modeling from an Object-Oriented Perspective

Henrik Nilsson[1], John Peterson[2], and Paul Hudak[3]

[1] School of Computer Science and IT, University of Nottingham, UK
nhn@cs.nott.ac.uk
[2] Math and Computer Information Science Department, Western State College, USA
jpeterson@western.edu
[3] Department of Computer Science, Yale University, USA
paul.hudak@yale.edu

Declaration: This paper is closely based on [19] that was published in the Proceedings of Practical Aspects of Declarative Languages (PADL) 2003. The paper has been updated and adapted for the Equation-Based Object-Oriented Languages and Tools (EOOLT) 2007 Workshop.

Abstract. The modeling and simulation of physical systems is of key importance in many areas of science and engineering, and thus can benefit from high-quality software tools. In previous research we have demonstrated how *functional programming* can form the basis of an expressive language for *causal* hybrid modeling and simulation. There is a growing realization, however, that a move toward *non-causal* modeling is necessary for coping with the ever increasing size and complexity of modeling problems. Our goal is to combine the strengths of functional programming and non-causal modeling to create a powerful, strongly typed *fully declarative modeling language* that provides modeling and simulation capabilities beyond the current state of the art: in particular, support for highly structurally dynamic systems. Additionally, we think our approach could serve as a semantical framework for studying modeling and simulation languages supporting structural dynamism, and maybe even as a core language in systems where the surface syntax is more conventional. Although our work is still in its very early stages, we believe that this paper clearly articulates the need for improved modeling languages and shows how functional programming techniques can play a pivotal role in meeting this need.

1 Introduction

Modeling and simulation is playing an increasingly important role in the design, analysis, and implementation of real-world systems. In particular, whereas modeling fragments of systems in isolation was deemed sufficient in the past, considering the interaction of these fragments *as a whole* is now necessary. The resulting models are large and complex, and span multiple physical domains.

Furthermore, these models are almost invariably *hybrid*: they exhibit both continuous-time and discrete-time behaviors. In fact, the very structure of the modeled system changes over time. Such models are known as *structurally dynamic*. In general, the total number of structural configurations, or *modes*, can be enormous, or even unbounded. We refer to systems whose number of modes cannot be practically predetermined as *highly structurally dynamic*. While supporting structural dynamism is hard, supporting highly structurally dynamic systems is even harder as this necessitates comprehensive and flexible solutions of a number of important subproblems: see Sect. 5.

There are two broad language categories of modeling and simulation languages. *Causal* (or *block-oriented*) languages are most popular; languages such as Simulink and Ptolemy II [13] represent this style of modeling. In causal modeling, the equations that represent the physics of the system must be written so that the direction of signal flow, the *causality*, is explicit. The second, but less populated, class of language is *non-causal*, where the model focuses on the interconnection of the components of the system being modeled, from which causality is then inferred. Such languages often support an *object-oriented* approach to modeling. Examples include Dymola [5] and Modelica [15].

The main drawback of causal languages is the need to explicitly specify the causality. This hampers modularity and reuse [2]. Non-causal languages address this problem by allowing the user to describe a model in a way which does not commit to any specific causality. The appropriate causality constraints are then inferred using both symbolic and numerical methods depending on how the model is being used. Unfortunately, current non-causal modeling languages tend to sacrifice generality when it comes to hybrid modeling: in particular, we are not aware of any *declarative* non-causal modeling language that supports highly structurally dynamic models, even if recent efforts like MOSILAB [20] and Sol [28] are important steps in that direction.

In previous research at Yale, we have developed a framework called *Functional Reactive Programming* (FRP) [26], which is suited for causal hybrid modeling. This framework is embodied in a language called *Yampa*.[4] as an extension of Haskell. Yampa permits highly structurally dynamic hybrid systems to be described clearly and concisely [18].[5] In addition, because the full power of a functional language is available, it exhibits a high degree of modularity, allowing reuse of components and design patterns. It also employs Haskell's polymorphic type system to ensure that signals are connected consistently, even as the system topology changes. The semantic foundations of Yampa are well defined and understood, making models expressed using Yampa suited for formal manipulation and reasoning. Yampa and its predecessors have been used in robotics simulation and control as well as a number of related domains [23, 24]. It has even been used for video games [4, 3]. We are currently investigating biological cell population modeling, where Yampa's support for highly structurally dynamic

[4] See `http://haskell.org/yampa`.

[5] However, at present, Yampa lacks integration with sophisticated numerical solvers, and its applicability for serious simulation work is thus limited

systems provides an interesting declarative approach to handling cell division in contrast to the imperative approach of agent-based simulators [12].

Non-causal modeling and FRP complement each other almost perfectly. We therefore aim to integrate the core ideas of FRP with non-causal modeling to create *Hydra*, a powerful, fully declarative modeling language combining the strengths of each. If we treat causality and dynamism as two dimensions in the modeling language design space, we see that Hydra occupies a unique point:

	Mostly static structure	Highly dynamic structure
Causal	Simulink	Yampa
Non-causal	Modelica	Hydra

MOSILAB and Sol are somewhere between Modelica and Hydra.

We refer to the combined paradigm of functional programming and non-causal, hybrid modeling as *Functional Hybrid Modeling*, or FHM. Conceptually, FHM can be seen as a generalization of FRP, since FRP's *functions* on signals are a special case of FHM's *relations* on signals. In its full generality, FHM, like FRP, also allows the description of structurally dynamic models.

The main contribution of this paper is that it outlines how notions appropriate for non-causal, hybrid simulation in the form of *first-class relations on signals* and *switch constructs* can be integrated into a functional language, yielding a non-causal modeling language supporting structural dynamism. It also identifies key research issues, and suggests how recent developments in the field of programming languages could be employed to address those issues.

2 Yampa

To help readers who are not familiar with Functional Reactive Programming put the ideas of this paper into context, we provide a brief review of the key ideas of Yampa in the following. For further details, see earlier papers on Yampa [9, 18]

2.1 Fundamental Concepts

Yampa is based on two central concepts: *signals* and *signal functions*. A signal is a function from time to a value:

$$Signal\ \alpha \approx Time \to \alpha$$

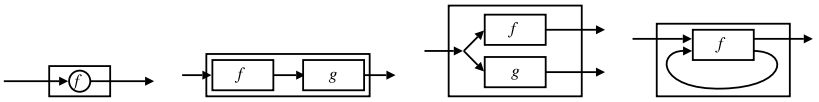
Time is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of a varying electrical voltage might be *Signal Voltage*.

A *signal function* is a function from *Signal* to *Signal*:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \to Signal\ \beta$$

When a value of type $SF\ \alpha\ \beta$ is applied to an input signal of type $Signal\ \alpha$, it produces an output signal of type $Signal\ \beta$. Signal functions are *first class entities* in Yampa. Signals, however, are not: they only exist indirectly through the notion of signal function. In general, the output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$. If a signal function is such that the output at time t only depends on the input at the very same time instant t, it is called *stateless*. Otherwise it is *stateful*.

2.2 Composing Signal Functions



(a) *arr f* (b) $f \ggg g$ (c) f &&& g (d) *loop f*

Fig. 1: Basic signal function combinators.

Programming in Yampa consists of defining signal functions compositionally using Yampa's library of primitive signal functions and a set of combinators. Yampa's signal functions are an instance of the arrow framework proposed by Hughes [10]. Three combinators from that framework are *arr*, which lifts an ordinary function to a stateless signal function, and the two signal function composition combinators $\lll$ and &&& :

$$
\begin{aligned}
&arr &&:: (a \to b) \to SF\ a\ b\\
&(\lll) &&:: SF\ b\ c \to SF\ a\ b \to SF\ a\ c\\
&(\ \&\&\&\ ) &&:: SF\ a\ b \to SF\ a\ c \to SF\ a\ (b, c)
\end{aligned}
$$

We can think of signals and signal functions using a simple flow chart analogy. Boxes represent signal functions, with one signal flowing in to the box's input port and another signal flowing out of the box's output port. Figure 1 illustrates some of the central arrow combinators using this analogy. The similarity to a block-oriented modeling language like Simulink is hopefully clear. The main difference is that the notion of composing blocks into larger blocks has been formalized through a handful of composition combinators, which is helpful from a semantical perspective, in contrast to the more unstructured approach of connecting outputs to inputs in an arbitrary fashion.

2.3 Arrow Syntax

While the arrow framework provides a useful semantical structure, it is not convenient for expressing large networks. It is much easier to simply connect whatever needs to be connected Simulink style, e.g. by naming nodes and then

explicitly stating the connection topology. Fortunately, it is easy to provide a layer of syntax that allows this, and then translate this into a network description in terms of the core arrow combinators. Paterson's arrow notation [22] does exactly that. An expression denoting a signal function has the form:

$$
\begin{array}{l}
\textbf{proc}\ pat \rightarrow \textbf{do} \\
\quad pat_1 \leftarrowtail sfexp_1 \prec exp_1 \\
\quad pat_2 \leftarrowtail sfexp_2 \prec exp_2 \\
\quad \ldots \\
\quad pat_n \leftarrowtail sfexp_n \prec exp_n \\
\quad returnA \prec exp
\end{array}
$$

The keyword **proc** is analogous to the λ in λ-expressions, pat and pat_i are patterns binding signal variables pointwise by matching on instantaneous signal values, exp and exp_i are expressions defining instantaneous signal values, and $sfexp_i$ are expressions denoting signal functions. The idea is that the signal being defined pointwise by each exp_i is fed into the corresponding signal function $sfexp_i$, whose output is bound pointwise in pat_i. The overall input to the signal function denoted by the **proc**-expression is bound by pat, and its output signal is defined by the expression exp. The signal variables bound in the patterns may occur in the signal value expressions, but *not* in the signal function expressions $sfexp_i$). An optional keyword **rec**, applied to a group of definitions, permits signal variables to occur in expressions that textually precede the definition of the variable, allowing recursive definitions (feedback loops).

For a concrete example, consider the following:

$$
\begin{array}{l}
sf = \textbf{proc}\ (a, b) \rightarrow \textbf{do} \\
\quad (c1, c2) \leftarrowtail sf1 \mathbin{\&\&\&} sf2 \prec a \\
\quad d \qquad\quad \leftarrowtail sf3 \lll sf4 \prec (c1, b) \\
\quad \textbf{rec} \\
\quad\quad e \leftarrowtail sf5 \prec (c2, d, e) \\
\quad returnA \prec (d, e)
\end{array}
$$

Note the use of the tuple pattern for splitting sf's input into two "named signals", a and b. Also note the use of tuple expressions and patterns for pairing and splitting signals in the body of the definition; for example, for splitting the output from $sf1 \mathbin{\&\&\&} sf2$. Also note how the arrow notation may be freely mixed with the use of basic arrow combinators.

2.4 Events

While some aspects of a program are naturally modeled as continuous signals, other aspects are more naturally modeled as *discrete events*. To this end, Yampa introduces the *Event* type, isomorphic to Haskell's *Maybe* type:

$$
\textbf{data}\ Event\ a = NoEvent \mid Event\ a
$$

The instantaneous value of signal of type *Event* T for some type T is either *NoEvent* or *Event* x for some value x of type T, thus mimicking a discrete-time signal that is only defined at discrete points in time.

2.5 Switching

The structure of a Yampa system may evolve over time. These structural changes are known as *mode switches*. This is accomplished through a family of *switching* primitives that use events to trigger changes in the connectivity of a system. The simplest such primitive is *switch*:

$$switch :: SF\ a\ (b, Event\ c) \rightarrow (c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

The *switch* combinator switches from one subordinate signal function into another when a switching event occurs. Its first argument is the signal function that initially is active. It outputs a pair of signals. The first defines the overall output while the initial signal function is active. The second signal carries the event that will cause the switch to take place. Once the switching event occurs, *switch* applies its second argument to the value of the event and switches into the resulting signal function.

Thus, note that the second argument of *switch* is a *function* of type $c \rightarrow SF\ a\ b$, that, when given the value of type c carried by the event, *dynamically computes* a new signal function to switch into. Using a Simulink analogy, *switch* in principle rips out a block, and then dynamically instantiates a parameterized block as a replacement. The design of *switch* thus exploits the fact that signal functions ("blocks") are first class entities in Yampa.

Yampa also includes *parallel* switching constructs that maintain *dynamic collections* of signal functions connected in parallel [18]. Signal functions can be added to or removed from such a collection at runtime in response to events, while *preserving* any internal state of all other signal functions in the collection; see Fig. 2. The first class status of signal functions in combination with switching over dynamic collections of signal functions makes Yampa an unusually flexible language for describing hybrid systems. For example, this makes it possible to handle systems where the number of modeled entities varies over time, like cell population models as mentioned earlier (Sect. 1).

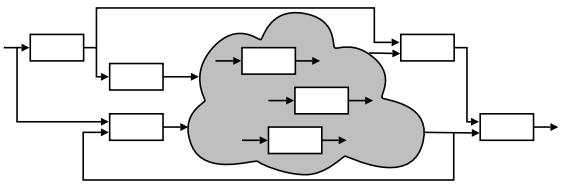
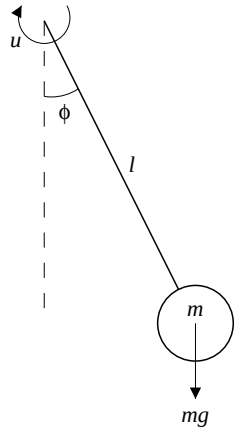


Fig. 2: System of interconnected signal functions with varying structure



(a) Pendulum

```
model BreakingPendulum
    parameter Real m=1, g=9.81, L=0.5;
    parameter Boolean Broken;
    input Real u;
    Real pos[2], vel[2];
    Real phi(start=PI/4), phid;
equation
    vel = der(pos);
    if not Broken then
        // Equations of pendulum.
        pos = {L*sin(phi), -L*cos(phi)};
        phid = der(phi);
        m*L*L*der(phid) + m*g*L*sin(phi) = u;
    else
        // Equations of free-flying mass.
        m*der(vel) = m*{0, -g};
    end if;
end BreakingPendulum;
```

(b) Modelica model

Fig. 3: A pendulum, subject to externally applied torque and gravity.

3 Non-Causal and Hybrid Modeling

While the simulation of pure continuous systems is relatively well understood, hybrid systems pose a number of unique challenges [16, 1]. Problems include handling a large number of modes, event detection, and consistent initialization of state variables. The integration of hybrid modeling with non-causal modeling raises further problems. Indeed, current non-causal modeling languages are quite limited in their ability to express hybrid systems. Many of the limitations are related to the symbolic and numerical methods that must be used in the non-causal approach. But another important reason is that most such systems insist on performing all symbolic manipulations *before* simulation begins [16]. Avoiding these limitations is an important part of our approach, see Sec. 5.

Since Modelica is representative of state-of-the-art, non-causal, hybrid modeling languages, we illustrate the limitations of present languages with an example from the Modelica documentation [14, pp. 31–33]. The system is a pendulum in the form of a mass m at the end of a rigid, mass-less rod, subject to gravity mg and an externally applied torque u at the point of suspension; see Fig. 3(a). Additionally, the rod could break at some point, causing the mass to fall freely.

Figure 3(b) shows a Modelica model of this system that, on the surface, looks like it achieves the desired result. Note that it has two modes, described by conditional equations. In the non-broken mode, the position `pos` and velocity `vel` of the mass are calculated from the state variables `phi` and `phid`. In the broken mode, `pos` and `vel` become the new state variables. This implies that state information has to be transferred between the non-broken and broken mode. Furthermore, the causality of the system is different in the two modes. When

non-broken, the equation relating `vel` and `pos` is used to compute `vel` from `pos`. When broken, the situation is reversed.

These facts make simulation hard. Modelica attempts to simplify matters by avoiding too radical structural changes. To that end, Modelica either requires the condition for selecting between two sets of equation to be a *parameter*, and thus unchanging during simulation, or else that the number of equations in each set are the same. In this case, as the number of equations is not the same, `Broken` has to be declared a parameter. Therefore the model above does not really solve the hybrid simulation problem at all! In order to actually model a pendulum that dynamically breaks at some point in time, the model must be expressed in some other way. The Modelica documentation suggests a causal, block-oriented formulation with explicit state transfer. Unsurprisingly, the result is considerably more verbose, nullifying the advantage of working in a non-causal language.

Moreover, even if `Broken` were allowed to be a dynamic variable, a fundamental problem would remain: once the pendulum has broken, it cannot become whole again. Modelica provides no way to declaratively express the *irreversibility* of this structural change. The best that can be done is to capture this fact indirectly through a state machine model that control the value of `Broken`.

4 Integrating Functional Programming and Non-Causal Modeling

In the preceding sections we discussed the advantages of non-causal modeling and the importance of hybrid modeling. We also pointed out serious shortcomings in current modeling languages with respect to these features. In this section, we describe a new way to combine non-causal and hybrid modeling techniques that addresses these issues. Inspired by FRP and Yampa, the two key ideas are to give first-class status to relations on signals and to provide constructs for discrete switching between relations. The result is Hydra, a functional hybrid modeling language capable of representing structurally dynamic systems.

While we, based on our experience of Yampa, believe that a language like Hydra would be a very expressive and powerful modeling and simulation language in its own right, we would like to emphasize that we also think our approach could serve as a valuable semantical framework for general study of modeling and simulation languages that supports structural dynamism, and maybe even as a core language in systems where the surface syntax is more conventional. Thus, what is important in the following is not the syntax (which is tentative and likely lacking in many ways), but the underlying principles.

4.1 First-Class Signal Relations

A natural mathematical description of a continuous signal function is that of an ODE in explicit form. A function is just a special case of the more general concept of a *relation*. While functions usually are given a causal interpretation, relations are inherently non-causal. Differential Algebraic Equations (DAEs), which are

at the heart of non-causal modeling, express dependences among signals without imposing a causality on the signals in the relation. Thus it is natural to view the meaning of a DAE as a non-causal *signal relation*, just as the meaning of an ODE in explicit form can be seen as a causal signal function. Since signal functions and signal relations are closely connected, this view offers a clean way of integrating non-causal modeling into an Yampa-like setting.

In the following, first-class signal relations are made concrete by proposing a (tentative) system for integrating them into a polymorphically typed functional language. Signal functions are also useful, but since they are just relations with explicit causality, we need not consider them in detail in the following.

Similarly to the signal function type *SF* of Yampa (Sect. 2.1), we introduce the type *SR* α for a relation on a signal of type *Signal* α. Specific relations use a more refined type; e.g., for the derivative relation *der* we have the typing:

$$der :: SR\ (Real, Real)$$

Since a signal carrying pairs is isomorphic to a pair of signals, we can understand *der* as a binary relation on two real-valued signals.

Next we need a notation for defining relations. Inspired by the arrow notation (Sect. 2.3), we introduce the following to denote a signal relation:

sigrel *pattern* **where** *equations*

The pattern introduces *signal variables* that at each point in time are bound to the *instantaneous* value of the corresponding signal. Given $p :: t$, we have:

sigrel p **where** $\ldots :: SR\ t$

Consequently, the equations express relationships between instantaneous signal values. This resembles the standard notation for differential equations in mathematics. For example, consider $x' = f(y)$, which means that the instantaneous value of the derivative of (the signal) x at every time instant is equal to the value obtained by applying the function f to the instantaneous value of y.

We introduce two styles of equations:

$$e_1 = e_2$$
$$sr \diamond e_3$$

where e_i are expressions (possibly introducing new signal variables), and sr is an expression denoting a signal relation. We require equations to be well-typed. Given $e_i :: t_i$, this is the case iff $t_1 = t_2$ and $sr :: SR\ t_3$.

The first kind of equation requires the values of the two expressions to be equal at all points in time. For example:

$$f(x) = g(y)$$

where f and g are functions.

The second kind allows an arbitrary relation to be used to enforce a relationship between signals. The symbol $\diamond$ can be thought of as *relation application*;

the result is a constraint which must hold at all times. The first kind of equation is just a special case of the second in that it can be seen as the application of the identity relation.

For another example, consider a differential equation like $x' = f(x, y)$. Using our notation, this equation could be written:

$$der \diamond (x, f(x, y))$$

where *der* is the relation relating a signal to its derivative. For convenience, a notation closer to the mathematical tradition should be supported as well:

$$\mathbf{der}(x) = f(x, y)$$

The meaning is exactly as in the first version.

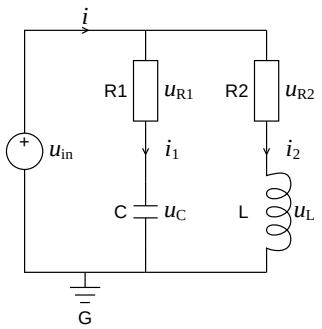


Fig. 4: A simple electrical circuit.

We illustrate our language by modeling the electrical circuit in Fig. 4.1 (adapted from [14]). The type *Pin* is a record type describing an electrical connection. It has fields v for voltage and i for current.[6]

$$\begin{aligned}
&\mathit{twoPin} :: \mathit{SR}\ (\mathit{Pin},\ \mathit{Pin},\ \mathit{Voltage})\\
&\mathit{twoPin} = \mathbf{sigrel}\ (p, n, u)\ \mathbf{where}\\
&\qquad u = p.v - n.v\\
&\qquad p.i + n.i = 0
\end{aligned}$$

$$\begin{aligned}
&\mathit{resistor} :: \mathit{Resistance} \to \mathit{SR}\ (\mathit{Pin},\ \mathit{Pin})\\
&\mathit{resistor}(r) = \mathbf{sigrel}\ (p, n)\ \mathbf{where}\\
&\qquad \mathit{twoPin} \diamond (p, n, u)\\
&\qquad r \cdot p.i = u
\end{aligned}$$

[6] The name *Pin* is perhaps a bit misleading since it just represents a pair of physical quantities, *not* a physical "pin component"; i.e., *Pin* is the type of *signal variables* rather than *signal relations*.

$$
\begin{aligned}
&\mathit{inductor} :: \mathit{Inductance} \to \mathit{SR}\ (\mathit{Pin},\ \mathit{Pin})\\
&\mathit{inductor}(l) = \textbf{sigrel}\ (p, n)\ \textbf{where}\\
&\qquad\qquad \mathit{twoPin} \diamond (p, n, u)\\
&\qquad\qquad l \cdot \textbf{der}(p.i) = u
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{capacitor} :: \mathit{Capacitance} \to \mathit{SR}\ (\mathit{Pin},\ \mathit{Pin})\\
&\mathit{capacitor}(c) = \textbf{sigrel}\ (p, n)\ \textbf{where}\\
&\qquad\qquad \mathit{twoPin} \diamond (p, n, u)\\
&\qquad\qquad c \cdot \textbf{der}(u) = p.i
\end{aligned}
$$

As in Modelica, the resistor, inductor and capacitor models are defined as extensions of the *twoPin* model. However, we accomplish this directly with functional abstraction rather than the Modelica class concept. Note how parameterized models are defined through functions *returning* relations. Since the parameters are normal function arguments, *not* signal variables, their values remain unchanged throughout the lifetime of the returned relations.[7]

To assemble these components into the full model, we adopt a Modelica-like **connect**-notation as a convenient abbreviation for connection equations. This is syntactic sugar which is expanded to proper connection equations, i.e. equality constraints or sum-to-zero equations depending on what kind of physical quantity is being connected. We assume that a voltage source model *vSourceAC* and a ground model *ground* are available in addition to the component models defined above. Moreover, we are only interested in the total current through the circuit, and, as there are no inputs, the model thus becomes a *unary* relation:

$$
\begin{aligned}
&\mathit{simpleCircuit} :: \mathit{SR}\ \mathit{Current}\\
&\mathit{simpleCircuit} = \textbf{sigrel}\ \mathrm{i}\ \textbf{where}\\
&\qquad\qquad \mathit{resistor}(1000) \diamond (\mathit{r1p}, \mathit{r1n})\\
&\qquad\qquad \mathit{resistor}(2200) \diamond (\mathit{r2p}, \mathit{r2n})\\
&\qquad\qquad \mathit{capacitor}(0.00047) \diamond (\mathit{cp}, \mathit{cn})\\
&\qquad\qquad \mathit{inductor}(0.01) \diamond (\mathit{lp}, \mathit{ln})\\
&\qquad\qquad \mathit{vSourceAC}(12) \diamond (\mathit{acp}, \mathit{acn})\\
&\qquad\qquad \mathit{ground} \diamond \mathit{gp}\\
&\qquad\qquad \textbf{connect}\ \mathit{acp},\ r1p,\ r2p\\
&\qquad\qquad \textbf{connect}\ r1n,\ \mathit{cp}\\
&\qquad\qquad \textbf{connect}\ r2n,\ \mathit{lp}\\
&\qquad\qquad \textbf{connect}\ \mathit{acn},\ \mathit{cn},\ \mathit{ln},\ \mathit{gp}\\
&\qquad\qquad i = \mathit{r1p}.i + \mathit{r2p}.i
\end{aligned}
$$

4.2 Modeling Systems with Dynamic Structure

In order to describe structurally dynamic systems we need to represent an evolving structure. To this end, we introduce two Yampa-inspired switching constructs: the *recurring switch* and the *progressing switch*. The recurring switch

[7] Compare to Modelica's **parameter**-variables mentioned in Sect. 3.

allows repeated switching between equation groups. In contrast, the progressing switch expresses that one group of equations *first* is in force, and then, *once* the switching condition has been fulfilled, another group, thus irreversibly progressing to a new structural configuration. For either sort of switching, difficult issues such as state transfer and proper initialization have to be considered.

We revisit the breaking pendulum example from Sect. 3 to illustrate these switching constructs. To deal with initialization and state transfer, we introduce special initialization equations that are only active at the time of switching, that is, during *events*, and we allow such equations to refer to the values of signal variables just prior to the event through a special **pre**-construct devised for that purpose. The initialization equations describe the initial conditions of the DAE after a switch. Mathematically, these equations must yield an initial value for every state variable in the new continuous equations. It is important that each branch of a switch can be associated with its own initialization equations, since each such branch may introduce its proper set of state variables. Initialization equations typically state continuity assumptions, like *pos* and *vel* below.

First, consider a direct transliteration of the equation part of the Modelica model using a recurring switch. The necessary initialization equations have also been added:

$$
\begin{array}{l}
vel = \mathbf{der}(pos) \\
\mathbf{switch}\ broken \\
\quad \mathbf{when}\ False\ \mathbf{then} \\
\quad\quad \mathbf{init}\ phi = pi/4 \\
\quad\quad \mathbf{init}\ phid = 0 \\
\quad\quad pos = \{l \cdot \sin(phi), -l \cdot \cos(phi)\} \\
\quad\quad phid = \mathbf{der}(phi) \\
\quad\quad m \cdot l \cdot l \cdot \mathbf{der}(phid) + m \cdot g \cdot l \cdot \sin(phi) = u \\
\quad \mathbf{when}\ True\ \mathbf{then} \\
\quad\quad \mathbf{init}\ vel = \mathbf{pre}(vel) \\
\quad\quad \mathbf{init}\ pos = \mathbf{pre}(pos) \\
\quad\quad m \cdot \mathbf{der}(vel) = m \cdot \{0, -g\}
\end{array}
$$

A recurring switch has one or more **when**-branches. The idea is that the equations in a **when**-branch are in force whenever the pattern after **when** (which may bind variables) matches the value of the expression after **switch**. Thus, whenever that value changes, we have an event and a switch occurs (this is similar to **case** in a functional language).

To express the fact that the pendulum cannot become whole once it has broken, we refine the model by changing to a progressing switch:

$$
\begin{array}{l}
vel = \mathbf{der}(pos) \\
\mathbf{switch}\ broken \\
\quad \mathbf{first} \\
\quad\quad \ldots \\
\quad \mathbf{once}\ True\ \mathbf{then} \\
\quad\quad \ldots
\end{array}
$$

A progressing switch has one **first**-branch and one or more **once**-branches. Initially, the equations in the **first**-branch are in force, but as soon as the value of the expression after **switch** matches one of the **once**-patterns, a switch occurs to the equations in the corresponding branch, after which no further switching occurs (for that particular instance of the switch).

By combining recursively-defined relations and progressing switches, it is possible to express very general sequences of structural changes over time, from simple mode transitions to making and breaking of connections between objects. A simple example of a recursively defined relation parameterized on a discrete state variable n is shown below. Initially, the relation behaves according to the equations in the **first**-branch, which may depend on n. Whenever the switching condition is fulfilled, the relation switches to a new instance of itself with the parameter n increased by one. In functional parlance, this is a form of tail call.

$$
\begin{aligned}
&\mathit{sysWithCntr} :: \mathit{Int} \rightarrow \mathit{SR}\ (\mathit{Real}, \mathit{Real}) \\
&\mathit{sysWithCntr}(n) = \textbf{sigrel}\ (x, y)\ \textbf{where} \\
&\qquad\qquad\qquad\qquad \textbf{switch}\ \ldots \\
&\qquad\qquad\qquad\qquad\quad \textbf{first} \\
&\qquad\qquad\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad\qquad\quad \textbf{once}\ \ldots\ \textbf{then} \\
&\qquad\qquad\qquad\qquad\qquad \mathit{sysWithCntr}(n+1) \diamond (x, y)
\end{aligned}
$$

As explained in Sect. 2.5, Yampa supports even more radical structural changes, including dynamic addition and deletion of objects. Our goal is to carry over as much as possible of that functionality to Hydra.

5 Implementation Issues

There are a number of challenges that must be addressed in an implementation of a language like Hydra. The primary issues are ensuring model correctness, simulation in the presence of dynamic mode changes, and mode initialization.

It is critical that dynamic changes in the model should should not weaken the static checking of the model, i.e. we want to ensure *compositional correctness*. A Haskell-like polymorphic type system, as in Yampa, ensures that the system integrity is preserved. In addition we would like to find at least necessary conditions for statically ensuring that causality analysis can always be carried out, that the equations at least could have a solution, and so on, regardless of how relations are composed dynamically. An example of a necessary but not sufficient condition is that the number of equations and number of variables agree, and that each variable can be paired with one equation. Since it will be necessary to keep track of the balance between equations and variables across relation boundaries, it is natural to integrate this aspect into the type system. Similar considerations apply to the number of initialization equations and continuous state variables. Recent work on dependent types is relevant here [27]. We also aim at extending the type system to handle physical dimensions [11].

In a highly structurally dynamic language, it is impossible to identify all possible operating modes and then factor them out as separate systems. Modes thus have to be generated *dynamically* during simulation as follows. Whenever a switch occurs, a new, global, "flattened" DAE has to be generated. The DAE is obtained by first carrying out the necessary discrete processing, which amounts to standard functional evaluation, including evaluation of the *relational expressions* in the equations that are to be active after the switch. The evaluation of relational expression is what creates *new instances* of relations, and carrying out the instantiation dynamically when switching occurs is what enables modeling of highly structurally dynamic systems. Once the new flattened DAE has been generated, it is subjected to causality analysis and other symbolic manipulations in preparation for simulation using suitable numerical methods [21, 6, 7]. The result is causal simulation code.

The hybrid bond graph simulator HyBrSim has demonstrated the feasibility of this dynamic approach, and that it indeed allows some difficult cases to be handled [17]. However, HyBrSim is an *interpreted* system. Simulation is thus slowed down both by occasional symbolic processing and by the interpretive overhead. To avoid interpretive overhead, we intend to leverage recent work on run-time code generation, such as 'C [8] or Cyclone [25]. We will need to adapt the sophisticated mathematical techniques used in existing non-causal modeling languages [21, 6, 7] to this setting. In part, it may be possible to do this systematically by *staging* the existing algorithms in a language like Cyclone.

The initial conditions of the (new) differential equations must be determined on transitions from one mode to another. However, arriving at consistent initial conditions is, in general, hard. Some state variables in the continuous part of the system may exhibit discontinuities at the time of switching while others will not: simply preserving the old value is not always the right solution. Structural changes could change the set of state variables, and the relationship between the new and old states may be difficult to determine. One approach is to require the modeler to provide a function that maps the old state to the new one for each possible mode transition [1]. However, the declarative formulation of non-causal models means that the simulator sometimes has a choice regarding which continuous variables should be treated as state variables. Requiring the user to provide a state mapping function is therefore not always reasonable.

A key to the success of HyBrSim is that bond graphs are based on physical notions such as energy and energy exchange, which are subject to continuity and conservation principles. We intend to generalize this idea by exploring the use of *declarations* for stating such principles, along the lines illustrated in Sec. 4.2. It may also be possible to infer continuity and conservation constraints automatically based on physical dimension types.

6 Related Work

There has been substantial interest in supporting structural dynamism within the non-causal modeling community recently. The most advanced effort at present

is probably MOSILAB [20]. Similarly to what is proposed here, MOSILAB supports dynamic addition and deletion of behavioral objects. The switching is controlled through a form statecharts. A modern, sophisticated DAE solver, with support for computing consistent initial conditions, is used.

However, the statechart approach implies an explicit enumeration of the modes, and even if the number of modes could be large due to combinatorial effects, this rules out a Yampa-style, truly dynamic number of simulation objects, which is the ultimate goal of Hydra.

Another aspect of MOSILAB is the use of Python for various meta-modeling tasks, such as writing "experiment scripts". We think that Hydra in itself, thanks to being a general-purpose functional language with first-class signal relations and functions, should be expressive enough to mostly provide equivalent meta-modeling capabilities, all in a uniform, declarative setting, without resorting to external imperative languages.

Sol [28] is another effort to create a non-causal modeling and simulation language supporting structural dynamism. It expressively avoids the statechart approach to retain more of the declarative clarity of languages like Modelica. It is also claimed that the Sol approach to dynamism scales better. A key aspect of the Sol approach is the capability to dynamically determine model instances. This idea seems to be somewhat similar to the notion of first-class signal functions and relations in Hydra. Like MOSILAB, Sol seems to stop short of the Hydra goal of supporting systems with a dynamic number of objects.

7 Conclusions

Hybrid modeling is a domain in which the techniques of declarative programming languages have the potential to greatly advance the state of the art. The modeling community has traditionally been concerned more with the mathematics of modeling than language issues. As a result, present modeling languages do not scale in a number of ways, particularly in hybrid systems that undergo significant structural changes. Hydra uses functional programming techniques to describe dynamically changing systems in a way that preserves the non-causal structure of the system specification and allows arbitrary switching among modes, yielding expressive power beyond current non-causal modeling languages.

Although we have not completed an implementation of Hydra, this paper demonstrates our basic design approach and maps out the design landscape. We expect that further research into the links between declarative languages and hybrid modeling will produce significant advances in this field.

Acknowledgments The authors would like to thank the anonymous EOOLT reviewers for many useful suggestions for adapting the paper to this venue.

References

1. Paul I. Barton and Cha Kun Lee. Modeling, simulation, sensitivity analysis, and optimization of hybrid systems. Submitted to ACM Transactions on Modelling

and Computer Simulation: Special Issue on Multi-Paradigm Modeling, September 2001.

2. François E. Cellier. Object-oriented modelling: Means for dealing with system complexity. In *Proceedings of the 15th Benelux Meeting on Systems and Control, Mierlo, The Netherlands*, pages 53–64, 1996.
3. Mun Hon Cheong. Functional programming and 3D games. BEng thesis, University of New South Wales, Sydney, Australia, November 2005.
4. Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.
5. Hilding Elmqvist, François E. Cellier, and Martin Otter. Object-oriented modeling of hybrid systems. In *Proceedings of ESS'93 European Simulation Symposium*, pages xxxi–xli, Delft, The Netherlands, 1993.
6. Hilding Elmqvist and Martin Otter. Methods for tearing systems of equations in object-oriented modeling. In *Proceedings of ESM'94, European Simulation Multiconference*, pages 326–332, Barcelona, Spain, June 1994.
7. Hilding Elmqvist, Martin Otter, and François E. Cellier. Inline integration: A new mixed symbolic/numeric approach. In *Proceedings of ESM'95, European Simulation Multiconference*, pages xxiii–xxxiv, Prague, Czech Republic, June 1995.
8. Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 131–144, January 1996.
9. Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.
10. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
11. Andrew Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, Computer Laboratory, April 1996. Published as Technical Report No. 391.
12. John King, Michael Lees, and Brian Logan. Agent-based and continuum modelling of populations of cells. Technical report, University of Nottingham, December 2006.
13. Edward A. Lee. Overview of the Ptolemy project. Technical memorandum UCB/ERLM01/11, Electronic Research Laboratory, University of California, Berkeley, March 2001.
14. The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial version 1.4*, December 2000. `http://www.modelica.org/documents/ModelicaTutorial14.pdf`.
15. The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification version 2.2*, February 2005. `http://www.modelica.org/documents/ModelicaSpec22.pdf`.
16. Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In Fritz W. Vaadrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control '99*, number 1569 in Lecture Notes in Computer Science, pages 165–177, 1999.
17. Pieter J. Mosterman, Gautam Biswas, and Martin Otter. Simulation of discontinuities in physical system models based on conservation principles. In *Proceedings of SCS Summer Conference 1998*, pages 320–325, July 1998.

18. Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.
19. Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.
20. 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, March 2005. Modelica Association.
21. Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, March 1988.
22. Ross Paterson. A new notation for arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, Firenze, Italy, September 2001.
23. John Peterson, Greg Hager, and Paul Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conference on Robotics and Automation*, May 1999.
24. John Peterson, Paul Hudak, Alastair Reid, and Greg Hager. FVision: A declarative language for visual tracking. In *Proceedings of PADL'01: 3rd International Workshop on Practical Aspects of Declarative Languages*, pages 304–321, January 2001.
25. Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for run-time code generation. Submitted for publication to JFP SAIG.
26. Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation*, pages 242–252, June 2000.
27. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.
28. Dirk Zimmer. Enhancing Modelica towards variable structure systems. In *Proceedings of 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT 2007)*, Berlin, Germany, July 2007. LiU E-Press.