Functional Hybrid Modeling

Henrik Nilsson, John Peterson, and Paul Hudak

Department of Computer Science, Yale University,
P.O. Box 208285 New Haven, CT 06520-8285 U.S.A.
{Henrik.Nilsson, John.C.Peterson, Paul.Hudak}@yale.edu

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. 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. For example, the modeled system may contain a digital controller, or it could be that the very structure of the modeled system changes over time. Either way, the resulting model will have a number of structural configurations, or *modes*, each described by continuous equations. In general, the total number of modes can be enormous, or even unbounded, and often cannot be predicted a priori. We refer to systems whose number of modes cannot be practically predetermined as *structurally dynamic.*

Special *modeling languages* have been developed to facilitate modeling and simulation. There are two broad language categories in this domain. *Causal* (or *block-oriented*) languages are most popular; languages such as Simulink [9] and Ptolemy II [8] 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* (or *object-oriented*[1]), where the model focuses on the interconnection of the components of the system being modeled, from which causality is then inferred. Examples include Dymola [3] and Modelica [11].

The main drawback of casual 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 avoid committing the model itself to a specific causality: depending on how the model is being used, the appropriate causality constraints are inferred using both symbolic and numerical methods. Unfortunately, current non-causal modeling languages sacrifice generality, particularly when it comes to hybrid modeling.

There are additional weaknesses that are common to both types of language. Many languages are either untyped, or important invariants are only checked dynamically. No commercially available modeling system enforces the consistent use of physical dimensions. Also, the number of modes is usually limited, since this simplifies implementation by making it possible to generate simulation code for all modes at compile time [12].

In previous research at Yale, we have developed a framework called *functional reactive programming*, or FRP [20], which is highly suited for causal hybrid modeling. This framework is embodied in a language called *Yampa*[2] as an extension of Haskell. Yampa permits highly dynamic hybrid systems to be described clearly and concisely [14]. 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 [16–18].

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:

	Static structure	Dynamic structure
Causal	Simulink	Yampa
Non-causal	Modelica	Hydra

We refer to the combined paradigm of functional programming and non-causal, hybrid modeling as *functional hybrid modeling*, or FHM. Conceptually,

[1] Not to be confused with object-oriented *programming* languages. Concepts like classes and inheritance may be part of an object-oriented modeling language, but methods and imperative variables are not.

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

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.

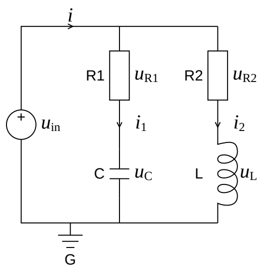
The main contribution of this paper is that it outlines how notions appropriate for non-causal, hybrid simulation in the form of *first-class relations on signals* and *switch constructs* can be integrated into a functional language, yielding a non-causal modeling language supporting structural dynamism. 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 Non-Causal and Hybrid Modeling

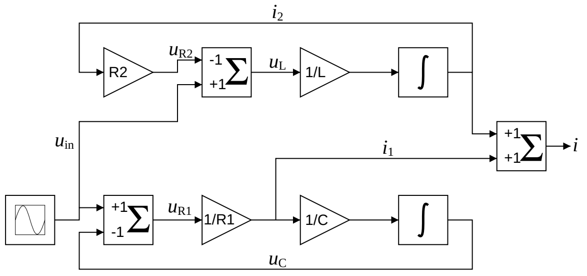
We believe that both the *non-causal* and *hybrid* styles of modeling are essential to address the increasing complexity of modeled systems. Unfortunately, combining these styles is difficult. In this section, we explain non-causal modeling and its advantages. We then outline the state of the art of non-causal hybrid modeling, and identify a number of shortcomings that must be addressed.

2.1 Advantages of Non-Causal Modeling

Consider the simple electrical circuit in Fig. 1(a) (adapted from [10]). We can model this circuit in a causal language such as Simulink by transforming the circuit into the *block diagram* of Fig. 1(b).



(a) Electrical circuit



(b) Causal model

Fig. 1: A simple electrical circuit and its causal model.

Mathematically, such a block diagram corresponds to a system of ordinary differential equations (ODEs) in explicit form.[3] In such an ODE, the *causality*,

[3] Since a system of ODEs can be written as a single ODE using vector notation, we will just write ODE and not worry about whether there are one or more equations.

i.e. the cause-and-effect relationship, is explicit: “known” quantities (inputs and state variables) are used to define “unknown” quantities (outputs and state derivatives). Hence the name causal modeling. The block diagram in Fig. 1(b) is a rendering of the following equations, where i_2 and u_C are the state variables:

$$\begin{array}{lll} u_{\mathrm{R}_2} = \mathrm{R}_2 i_2, & u_{\mathrm{L}} = u_{\mathrm{in}} - u_{\mathrm{R}_2}, & {i_2}' = \frac{u_{\mathrm{L}}}{\mathrm{L}} \\ u_{\mathrm{R}_1} = u_{\mathrm{in}} - u_{\mathrm{C}}, & i_1 = \frac{u_{\mathrm{R}_1}}{\mathrm{R}_1}, & {u_{\mathrm{C}}}' = \frac{i_1}{\mathrm{C}} \\ & i = i_1 + i_2 & \end{array}$$

With causal modeling, it is easy to derive the simulation code by transliterating the ODEs into a sequence of assignment statements that compute the outputs and the derivatives of the state variables for each time step. Simulation is then just a matter of stepwise numerical integration.

Unfortunately, the above equations, and consequently also the corresponding block diagram, bear little structural resemblance to the physical circuit they model. The burden of deriving the causal model rests entirely with the modeler, and is generally a difficult task. In particular, a causal model is not *compositional*: it cannot be expressed structurally as a composition of physical models of the individual components. For instance, consider a resistor. In a causal model, this component may be modeled (via Ohm’s law) by one of two equations, $u = \mathrm{R}i$, or $i = u/\mathrm{R}$, depending on whether the voltage needs to be computed from the current or vice versa. Thus, no single type of causal representation can capture the behavior of a resistor. Details of how the equations that define the model are to be solved dictate how the user must express the model. In practice, modeling using causal equations is quite “brittle”: a small change in the physical structure of the system may have global consequences in the causality of the equations. This make it difficult to reuse components in models [2].

In contrast, non-causal modeling frees the modeler from the need to spell out the “how” of the simulation code through an explicit ODE. A non-causal model is an implicit system of differential and algebraic equations (DAE):

$$\mathbf{f}(\mathbf{x}, \mathbf{x}', \mathbf{w}, \mathbf{u}, t) = \mathbf{0}$$

where $\mathbf{x}$ is a vector of state variables, $\mathbf{w}$ is a vector of algebraic variables, $\mathbf{u}$ is a vector of inputs, and t is the time. This allows the modeler to express the model in a way that directly reflects its physical structure. Models of individual components can be *reused* without first having to adapt them according to any specific causality requirements.

For example, a non-causal model of a resistor can be formulated as follows:

$$\begin{aligned} u &= v_{\mathrm{p}} - v_{\mathrm{n}} \\ i_{\mathrm{p}} + i_{\mathrm{n}} &= 0 \\ u &= \mathrm{R} i_{\mathrm{p}} \end{aligned}$$

where the subscripts p and n signify the positive and negative pin of the component, respectively. A non-causal model for an inductor is given by the following equations:

$$\begin{aligned} u &= v_{\mathrm{p}} - v_{\mathrm{n}} \\ i_{\mathrm{p}} + i_{\mathrm{n}} &= 0 \\ u &= \mathrm{L}{i_{\mathrm{p}}}' \end{aligned}$$

Note that the equations are identical to those in the resistor model, except for the last one. This is also the case for a capacitor model where the last equation would read:

$$i_{\mathrm{p}} = \mathrm{C}u'$$

In the context of a composite model, such as the circuit from Fig. 1(a), the models of individual components can be reused simply by copying the equations (and renaming variables to avoid name clashes). The sub-models are then interconnected by adding connection equations according to Kirchhoff's voltage and current laws. For instance, after suitable renaming, the connection equations for the node between the resistor R_1 and the capacitor C would be:

$$\begin{aligned} v_{\mathrm{R}_1,\mathrm{n}} &= v_{\mathrm{C},\mathrm{p}} \\ i_{\mathrm{R}_1,\mathrm{n}} + i_{\mathrm{C},\mathrm{p}} &= 0 \end{aligned}$$

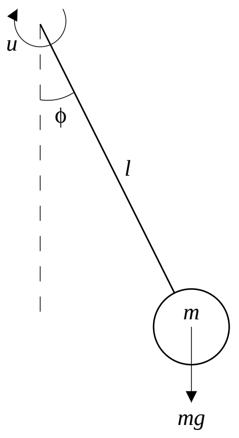
Good abstraction mechanisms can facilitate the mechanical aspects of copying code and creating connections, allowing the user to define component models in the form of named groups of equations and then create multiple interconnected instances by referring to the components by name. A non-causal language can also support component hierarchies, allowing the reuse of modeling knowledge in an *object-oriented* way. For example, the common aspects of the resistor, inductor and capacitor models above (the two first equations of each model) can be collected into a superclass describing what is common for two-pin components. This modeling knowledge would then be reused in the actual component models through inheritance from the common superclass. A good example of such a language is Modelica [11].

A non-causal simulation tool must undertake a substantial amount of symbolic processing to put the model into a form suitable for simulation. While there are numerical methods for integrating implicit DAEs, these methods are not suitable for solving higher-index[4] DAEs which are very common in practice. Fortunately, it is possible to automatically reduce the index of a DAE to 1 through symbolic manipulations [15], and further transformations allow the system to be put into a form which can be solved efficiently by specialized numerical methods [4, 5].

2.2 The Need for Non-Causal Hybrid Modeling

A *hybrid model* contains both *continuous* and *discrete* values. The continuous and discrete parts of the model interact via discrete transitions at distinct points

[4] The *index* of a DAE is the number of symbolic differentiations it takes to transform the system to an ODE.



(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. 2: A pendulum, subject to externally applied torque and gravity.

in time. These interactions are known as *events*. In between events, the model evolves continuously: all discrete values remain fixed. Since the model may depend conditionally on the discrete values, each discrete value assignment defines a potentially unique configuration or *mode* of continuous operation.

While the simulation of pure continuous systems is relatively well understood, hybrid systems pose a number of unique challenges [12, 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 a more important reason is that most such systems insist on performing all symbolic manipulations *before* simulation begins [12]. Avoiding these limitations is an important part of our approach, see Sec. 4.

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 [10, 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. 2(a). Additionally, the rod could break at some point, causing the mass to fall freely.

Figure 2(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. So much so that Modelica does not handle these equations correctly, because it *forbids* conditional equations with dynamic conditions. Thus, `Broken` is declared to be a *parameter*, meaning that it will remain constant during simulation. 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.

Thus we see that even quite simple examples go beyond the non-causal modeling capabilities of one of the most advanced non-causal, hybrid modeling languages currently available. 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. However, 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 and use that to control the value of `Broken`. But this makes the resulting model harder to understand, and it is also difficult for a simulator to exploit the fact that a certain set of equations and variables cannot be used again (to save memory and computational resources) since the simulator would have to infer this fact from the state machine model.

3 Integrating Functional Programming and Non-Causal Modeling

In the previous section we pointed out 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. 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 declarative, semantically coherent, functional hybrid modeling language capable of representing structurally dynamic systems.

3.1 First-Class Signal Relations

A *signal* is, conceptually, a function of time. A *signal function* maps a stimulating signal onto a responding signal; i.e., a signal function is just a (causal) block in the terminology of block-oriented modeling languages. A natural mathematical description of a continuous signal function is that of an ODE in explicit form. Signal functions are first-class entities in Yampa: they have a type, they can be bound to variables, they can be passed to and returned from functions. This is the key to the tight integration of the discrete and continuous aspects of Yampa, and is what makes Yampa uniquely flexible as a language for hybrid modeling.

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. 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, which is the essence of Hydra.

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.

Conceptually, we define the polymorphic type of signals as $\texttt{S}\ \alpha = \texttt{Time} \rightarrow \alpha$; that is, $\texttt{S}\ \alpha$ is the type of a signal whose instantaneous value is of type α. However, signals only exist implicitly via signal functions and signal relations: there is no syntactic entity which has type $\texttt{S}\ \alpha$. We then introduce the type

$\texttt{SR}\ \alpha$

for a relation on a signal of type $\texttt{S}\ \alpha$. Specific relations use a more refined type. For example, for the derivative relation *der* we have the typing:

$\mathit{der} :: \texttt{SR}\ (\texttt{Real}, \texttt{Real})$

where :: is the typing relation. 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 notation for defining relations. The following construct, in spirit analogous to a λ-abstraction, denotes a signal relation:

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

The pattern introduces *signal variables* which at each point in time are bound to the *instantaneous* value (a "sample") of the corresponding signal. Thus, given $p :: t$, we have:

sigrel p **where** $\ldots$ $::$ $\texttt{SR}\ t$

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 $sr :: \texttt{SR}\ t_3$.

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 a special case of the second in the following sense: if taking the syntactic liberty to allow $=$ to denote the identity relation ($= :: \texttt{SR}\ (\texttt{Real},\texttt{Real})$), one could write $f(x) = g(x)$ as

$$= \diamond (f(x), g(x))$$

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.

We illustrate our language by modeling the electrical circuit from Fig. 1(a). The type `Pin` is a record type describing an electrical connection. It has fields v for voltage and i for current.[5]

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

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

[5] 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*.

```
*inductor* :: Inductance → SR (Pin, Pin)
*inductor*(l) = **sigrel** (p, n) **where**
                    twoPin ◇ (p, n, u)
                    l · **der**(p.i) = u

*capacitor* :: Capacitance → SR (Pin, Pin)
*capacitor*(c) = **sigrel** (p, n) **where**
                    twoPin ◇ (p, n, u)
                    c · **der**(u) = p.i
```

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. With first-class relations we have a language that is both simpler and more expressive. 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.[6]

To assemble these components into the full model, we will 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:

```
simpleCircuit :: SR Current
simpleCircuit = sigrel i where
                    resistor(1000) ◇ (r1p, r1n)
                    resistor(2200) ◇ (r2p, r2n)
                    capacitor(0.00047) ◇ (cp, cn)
                    inductor(0.01) ◇ (lp, ln)
                    vSourceAC(12) ◇ (acp, acn)
                    ground ◇ gp
                    connect acp, r1p, r2p
                    connect r1n, cp
                    connect r2n, lp
                    connect acn, cn, ln, gp
                    i = r1p.i + r2p.i
```

3.2 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

[6] Compare to Modelica's **parameter**-variables mentioned in Sec. 2.2.

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 will revisit the breaking pendulum example from Sec. 2.2 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, as in the case of *pos* and *vel* below.[7]

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:

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

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:

```
$vel = \mathbf{der}(pos)$
```

[7] Since Modelica does not support hybrid models where the set of state variables changes, it does not provide any declarative constructs for relating the states across modes. However, it does provide an essentially imperative construct for reinitializing individual state variables.

```
switch broken
  first
    ...
  once True then
    ...
```

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.

```
sysWithCntr :: Int → SR (Real, Real)
sysWithCntr(n) = sigrel (x, y) where
                   switch ...
                     first
                       ...
                     once ... then
                       sysWithCntr(n + 1) ◇ (x, y)
```

Yampa supports even more radical structural changes, including dynamic addition and deletion of objects [14]. We hope to carry over much of that functionality to Hydra as well.

4 Implementation Issues

There are a number of significant 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. The static and dynamic semantics of the language must also be worked out in detail. The dynamic semantics are best described using a reference implementation as an embedding in a functional language such as Haskell. Good surface syntax is also important and can be provided through a pre-processor.

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*. Using a Haskell-like polymorphic type system, as in FRP, 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 [21]. We also aim at extending the type system to handle physical dimensions [7].

In a structurally dynamic language, it will be impossible to identify all possible operating modes and then factor them out as separate systems. We intend to generate the modes *dynamically* during simulation. In non-causal modeling, that implies that causality analysis and the prerequisite symbolic processing has to be performed whenever mode switches occur during simulation. The hybrid bond graph simulator HyBrSim has demonstrated the feasibility of this approach, and that it indeed allows some difficult cases to be handled [13]. 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 [6] or Cyclone [19]. We will need to adapt the sophisticated mathematical techniques used in existing non-causal modeling languages [15, 4, 5] to this setting. In part, it may be possible to do this systematically by *staging* the existing algorithms in a language like Cyclone.

Whenever a switch occurs, a new, global, “flattened” DAE has to be generated. This DAE is what governs the overall continuous system behavior until the next discrete event. It is obtained by first carrying out the necessary discrete processing. This 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 truly 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 [15, 4, 5]. The result is causal simulation code (a sequence of “assignment statements”), which should be compiled dynamically for better efficiency.

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. 3.2. It may also be possible to infer continuity and conservation constraints automatically based on physical dimension types.

Nevertheless, particularly when dealing with systems with highly dynamic structure, manual intervention may be necessary. In our work on Yampa, we have developed high-level mechanisms that exploit the first-class status of signal functions to give the user fine control over state transfer across mode switches [14]. We hope to generalize these results to signal relations and a non-causal setting in Hydra.

5 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.

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. Franois 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. Hilding Elmqvist, Franois 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.
4. 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.

5. Hilding Elmqvist, Martin Otter, and Franois 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.
6. 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.
7. Andrew Kennedy. *Programming Languages and Dimensions.* PhD thesis, University of Cambridge, Computer Laboratory, April 1996. Published as Technical Report No. 391.
8. Edward A. Lee. Overview of the ptolemy project. Technical memorandum UCB/ERLM01/11, Electronic Research Laboratory, University of California, Berkeley, March 2001.
9. The MathWorks, Inc. *Using Simulink Version 4*, June 2001.
10. The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial version 1.4*, December 2000.
11. The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification version 2.0*, July 2002.
12. 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.
13. 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.
14. 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.
15. Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, March 1988.
16. Izzet Pembeci, Henrik Nilsson, and Greogory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Principles and Practice of Declarative Programming (PPDP'02)*, Pittsburgh, Pennsylvania, USA, October 2002.
17. John Peterson, Greg Hager, and Paul Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conference on Robotics and Automation*, May 1999.
18. 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.
19. Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for run-time code generation. Submitted for publication to JFP SAIG.
20. 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.
21. 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.