Functional Programming Gets Physical: Pushing the Boundaries of Non-Causal Modelling Languages

Keynote Talk, DSL 2009, Oxford, 16 July 2009

Henrik Nilsson

School of Computer Science
University of Nottingham

Modelling and Simulation (1)

Developing models and studying their properties and behaviour are of immense theoretical and practical importance. Some examples:

- Science:
  - Weather forecasting
  - Biological cell models (e.g. neurons, neocortical column)
  - Galaxy formation
  - and many, many more . . .

Modelling and Simulation (2)

- Engineering, from nanotechnology to skyscrapers and space shuttles:
  - initial development
  - performance optimisation
  - safety engineering

Early Mechanical Simulation Efforts

British wooden mechanical horse simulator

Computer Modelling and Simulation

- Modelling and simulation has been a main application of digital computers from the start: Monte Carlo simulation of nuclear detonation (Manhattan project, Los Alamos)
- Recent examples:
  - Los Alamos molecular ribosome model: 2.64 million atoms.
  - The Blue Brain Project: Simulation of 10000-neuron, biologically accurate neocortical column on 8192-processor IBM Blue Gene super computer.

DSLs and Modelling (1)

The need for Domain-Specific languages to allow scientists and engineers to develop models is evident:

- Domain-experts are usually not programmers
- The scale of the problems is such that high-level, domain-specific notation and tools are essential to get the work done.

DSLs and Modelling (2)

Some examples:

- Spice (analogue circuits)
- VHDL-AMS (mixed digital/analogue circuits)
- NEURON (neuron modelling)
- gPROMS (process industries)
- Simulink (domain-neutral, continuous-time)
- Stateflow (event-driven simulation)
- Modelica (domain-neutral, hybrid)

DSLs and Modelling (3)

Quote from NEURON site (www.neuron.yale.edu):

> Instead of forcing users to reformulate their conceptual models to fit the requirements of a general purpose simulator, NEURON is designed to let them deal directly with familiar neuroscience concepts. Consequently, users can think in terms of the biophysical properties of membrane and cytoplasm, the branched architecture of neurons, and the effects of synaptic communication between cells.

Reflection

Modelling and simulation constitute a big, diverse, and very important application area with a clear need for DSLs. But it is also “crowded”: there are plenty of successful DSLs already, both

- multi-(modelling-)domain
- (modelling-)domain-specific

Question: What can Computer Science and specifically Programming Language Research contribute?

The Rest of This Talk

- Declarative, non-causal, hybrid modelling of physical systems:
  - electrical circuits
  - robot manipulators
  - chemical plants
  - . . .
- Modelica
- Functional Hybrid Modelling

Non-Causal, Hybrid Modelling?

- ***Declarative, non-causal***: models expressed as systems of undirected Differential Algebraic Equations (DAE)
- ***Multi-domain***: models spanning multiple physical domains
- ***Hybrid***: models exhibiting both continuous-time and discrete time behaviour; e.g., structural changes at discrete points in time:
  - model simplifications
  - structural changes
  - discrete subsystems

Causal vs. Non-Causal Modelling (1)

Causal or ***block-oriented*** modelling: model is ODE in ***explicit*** form:

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

Causality, i.e. cause-effect relationship, given by the modeller. Cf. Functional Programming.

Causal modelling is the dominating modelling paradigm; languages include Simulink.

Causal vs. Non-Causal Modelling (2)

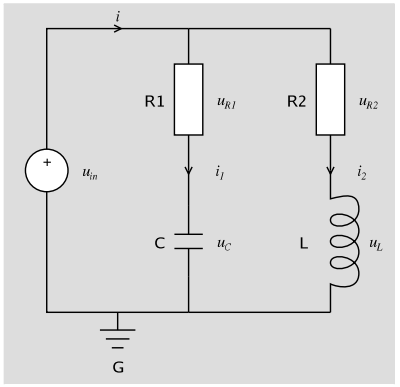
Non-causal or "***object-oriented***" modelling: model is DAE in implicit form:

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

Causality inferred by simulation tool from usage context. Cf. Logic Programming.

Non-causal modelling is a fairly recent development; languages include Dymola and Modelica.

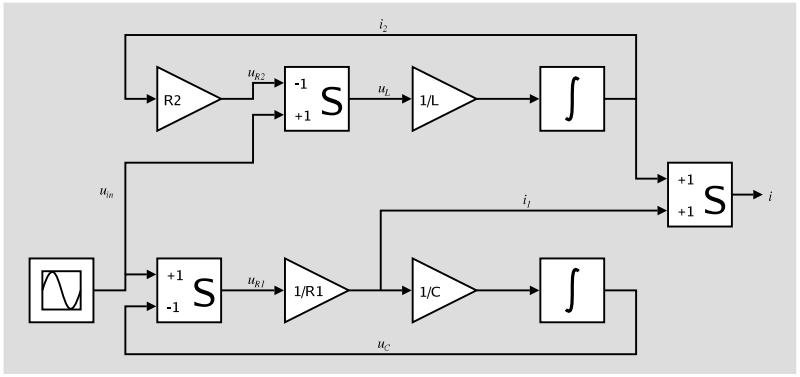
Causal Modelling: Example (1)



$$\begin{aligned} u_{R_2} &= R_2 i_2 \\ u_L &= u_{in} - u_{R_2} \\ {i_2}' &= \frac{u_L}{L} \\ u_{R_1} &= u_{in} - u_C \\ i_1 &= \frac{u_{R_1}}{R_1} \\ {u_C}' &= \frac{i_1}{C} \\ i &= i_1 + i_2 \end{aligned}$$

Causal Modelling: Example (2)

Or, as a block diagram:



Drawbacks of Causal Modelling

Causal modelling bad fit for fundamentally non-causal domains like physical modelling:

- Structure of model and modelled system does not agree.
- Model not simple composition of models of physical components.
- Fixed causality hampers reuse.
- Burden of deriving a non-causal model rests with the modeller.

Non-Causal Modelling: Example (1)

Non-causal resistor model:

$$\begin{aligned} u &= v_p - v_n \\ i_p + i_n &= 0 \\ u &= R i_p \end{aligned}$$

Non-causal inductor model:

$$\begin{aligned} u &= v_p - v_n \\ i_p + i_n &= 0 \\ u &= L i_p' \end{aligned}$$

Non-Causal Modelling: Example (2)

A non-causal model of the entire circuit is created by ***instantiating*** the component models: copy the equations and rename the variables.

The instantiated components are then ***composed*** by adding connection equations according to Kirchhoff's laws, e.g.:

$$\begin{aligned} v_{R_1,n} &= v_{C,p} \\ i_{R_1,n} + i_{C,p} &= 0 \end{aligned}$$

Very direct: can be accomplished through a drag-and-drop GUI.

Modelica (1)

Modelica:

- non-causal
- declarative
- domain-neutral; supports e.g.
  - mechanical
  - electrical
  - hydraulic
  - thermal

  modelling and multi-modelling.

Modelica (2)

- Being developed since late 1990s.
- Supported by a number of sophisticated implementations (e.g. from Dynasim (part of Dassault, using Modelica with CAD system CATIA), Maplesoft, ITI GmbH).
- Standard library with 780 generic model components, 550 functions.
- In widespread use; e.g. in automotive industry (Ford, GM, Toyota, . . .).

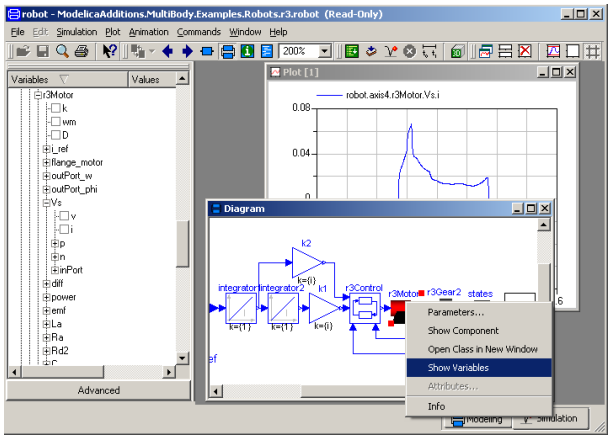
Modelica (3)

- ***Integrated*** language support for GUI modelling and simulation:
  - Models can be decorated with graphical annotations to specify the graphical view.
  - Ways to associate model parameters and variables with textual descriptions.

(However, not all language constructs have a graphical representation.)

Modelica (4)

Modelling and simulation through typical GUI



Simple Circuit in Modelica (1)

```
connector Pin
    Voltage v;
    flow Current i;
end Pin;

partial model TwoPin
    Pin p, n;
    Voltage u;
equation
    u = p.v - n.v;
    p.i + n.i = 0;
end TwoPin
```

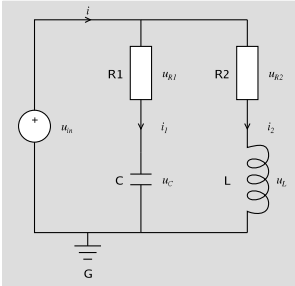

Simple Circuit in Modelica (2)

```
model Resistor
    extends TwoPin;
    parameter Resistance R;
equation
    R * p.i = u;
end Resistor;

model Inductor
    extends TwoPin;
    parameter Inductance L;
equation
    L * der(p.i) = u;
end Inductor;
```

Simple Circuit in Modelica (3)

```
model SimpleCircuit
    Resistor  R1(R=1000), R2(R=2200);
    Capacitor C(C=0.00047);
    Inductor  L(L=0.01);
    VsourceAC AC(AC=12);
    Ground G;
equation
    connect(AC.p, R1.p);
    connect(R1.n, C.p);
    connect(C.n,  AC.n);
    connect(R1.p, R2.p);
    connect(R2.n, L.p);
    connect(L.n,  C.n);
    connect(AC.n, G.p);
end;
```



Any Issues? (1)

- Modelica representative of many DSLs in being designed with strong input from domain experts, i.e. people with in depth understanding of
  - ***what the needs are***
  - ***how to meet those needs using current technology.***
- As a result, Modelica is a great language for "getting the job done".

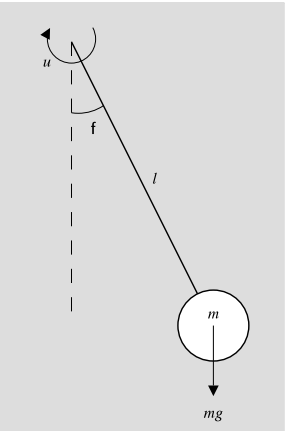
Any Issues? (2)

But, Modelica has recognised limitations, and, from a CS perspective, the Modelica design arguably cuts a few corners. For example:

- Limited hybrid modelling capabilities.
- "Declarative", but imperative origins show (both syntactically and semantically).
- Very complicated, rather ad-hoc type system.

I believe Programming Language Research can provide useful input to address some of these concerns.

The Breaking Pendulum (1)



Pendulum modelled as point mass fixed at the end of rigid, mass-less rod, subject to gravity and applied torque. Rod breaks at given point in time, allowing mass to fall freely.

The Breaking Pendulum (2)

```
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;
...
```


The Breaking Pendulum (3)

```
...
equation
   vel = der(pos);
   if not Broken then
      // Equations for pendulum.
      pos = {L*sin(phi), -L*cos(phi)};
      phid = der(phi);
      m*L*L*der(phid) + m*g*L*sin(phi) = u;
   else
      // Equations for mass falling freely.
      m*der(vel) = m*{0, -g};
   end if;
end BreakingPendulum;
```


The Breaking Pendulum (3)

Unfortunately, Modelica does ***not*** allow `Broken` to change ***during*** simulation in this case, as the implied structural changes are too radical. What are the reasons?

- Lots of hard simulation problems in the general case.
- Language design decision: Modelica designed to allow compilation of simulation code prior to simulation.

Quiz: Spot the Dependent Types!

(No prize, I'm afraid!)

- Modelica provides a rich sublanguage for expressing and operating on arrays.
- The number of dimensions and their sizes are considered to be part of the type.

```
function joinThreeVectors
  input Real v1[:],v2[:],v3[:];
  output Real v[size(v1,1)+size(v2,1)+size(v3,1)];
algorithm
  v := cat(1,v1,v2,v3);
end joinThreeVectors;
```

Array Checking in Modelica

- The Modelica standard promises very little:
  - Size constraints will be checked at some point.
  - When is a “quality-of-implementation issue”.
  - The common approach is to generate an assertion unless the involved sizes happen to be known statically.
- But at least the need for flexible array sizes and checking them for consistency is recognised.

An Opportunity?

An opportunity for using e.g. “proper” dependent types?

Possibly, but need to keep the end-users in mind:

- They are experts in their domains, not Computer Scientists.
- And they have a job to do.
- So unlikely to be impressed if asked to prove e.g.

  $n + m = m + n$ or $m(n + 1) = m + mn$

  (and might not know how).

Functional Hybrid Modelling (1)

Functional Hybrid Modelling (FHM):

- Novel approach to designing purely declarative languages for non-causal modelling and simulation.
- Vehicle for research into semantic foundations with a view to
  - improve hybrid capabilities
  - designing domain-specific type systems
- Modelling language in its own right, as well as a “back-end” for more traditional languages.

Functional Hybrid Modelling (2)

- FHM was inspired by Functional Reactive Programming, in particular Yampa.
- Initially joint work with Paul Hudak and John Peterson.
- Currently working with George Giorgidze.

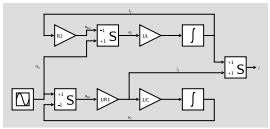
Yampa in a Nutshell (1)

- Yampa (in principle) supports ***causal*** modelling through functions on time-continuous ***signals***, so called ***Signal Functions***. Conceptually:

  $\texttt{Signal}\ \alpha \approx \texttt{Time} \rightarrow \alpha$

  $\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$

- Think block diagrams:



Yampa in a Nutshell (2)

- Yampa has two “layers”:
  - The ***functional layer***, or host language (Haskell)
  - The ***reactive layer***: signal functions and related constructs
- Signal Functions are ***first class entities*** in the functional layer. As a result:
  - (Some) meta modelling capabilities for free.
  - Very flexible hybrid modelling as new models can be ***computed*** dynamically, during simulation.

Yampa in a Nutshell (3)

- In typical hybrid fashion, execution alternates between:
  - event processing at discrete points in time, possibly resulting in structural reconfigurations (functional layer)
  - processing of (conceptually) continuous signals in between events (reactive layer).

Example: Space Invaders

Back to FHM

FHM attempts to combine the advantages of FRP with non-causal modelling:

- First-class ***relations*** on signals instead of functions on signals to enable non-causal modelling.
- Employ state-of-the-art symbolic and numerical methods for sound and efficient simulation.

Think of a ***signal relation*** as a (fragment of) a DAE system.

First class signal relations

The type for e.g. a binary signal relation:

$\texttt{SR}\ (\texttt{Real}, \texttt{Real})$

E.g. the derivative relation:

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

Defining Relations

The following construct denotes a signal relation:

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

The pattern introduces ***signal variables*** that at each point in time are going to be bound to to a "sample" of the corresponding signal.

Given $p :: t$, we have:

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

Equations

Let $e_i :: t_i$ be non-relational expressions possibly introducing new signal variables.

Point-wise equality; the equality must hold for all points in time:

$e_1 = e_2$

Relation ***application***; the relation must hold for all points in time:

$sr \diamond e_3$

Here, sr is an ***expression*** having type $\texttt{SR}\ t_3$.

Modelling Electrical Components (1)

The type `Pin` is assumed to be a record type describing an electrical connection. It has fields v for voltage and i for current.

$$twoPin :: \texttt{SR}\ (\texttt{Pin}, \texttt{Pin}, \texttt{Voltage})$$
$$twoPin = \textbf{sigrel}\ (p, n, v)\ \textbf{where}$$
$$v = p.v - n.v$$
$$p.i + n.i = 0$$

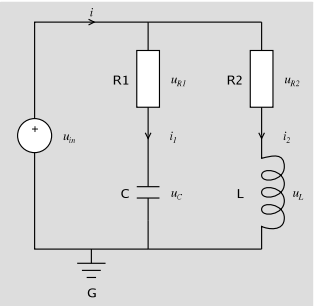
Modelling Electrical Components (2)

$$resistor :: \texttt{Resistance} \rightarrow \texttt{SR}\ (\texttt{Pin}, \texttt{Pin})$$
$$resistor\ r = \textbf{sigrel}\ (p, n)\ \textbf{where}$$
$$twoPin \diamond (p, n, v)$$
$$r \cdot p.i = v$$
$$inductor :: \texttt{Inductance} \rightarrow \texttt{SR}\ (\texttt{Pin}, \texttt{Pin})$$
$$inductor\ l = \textbf{sigrel}\ (p, n)\ \textbf{where}$$
$$twoPin \diamond (p, n, v)$$
$$l \cdot \textbf{der}(p.i) = v$$

Modelling an Electrical Circuit (1)

$$simpleCircuit :: \texttt{SR Current}$$
$$simpleCircuit = \textbf{sigrel}\ \mathsf{i}\ \textbf{where}$$
$$resistor(1000) \diamond (r1p, r1n)$$
$$resistor(2200) \diamond (r2p, r2n)$$
$$capacitor(0.00047) \diamond (cp, cn)$$
$$inductor(0.01) \diamond (lp, ln)$$
$$vSourceAC(12) \diamond (acp, acn)$$
$$ground \diamond gp$$
$$\ldots$$

Modelling an Electrical Circuit (2)



$$\ldots$$
$$\textbf{connect}\ acp, r1p, r2p$$
$$\textbf{connect}\ r1n, cp$$
$$\textbf{connect}\ r2n, lp$$
$$\textbf{connect}\ acn, cn, ln, gp$$
$$i = r1p.i + r2p.i$$

Prototype Hydra Implementation (1)

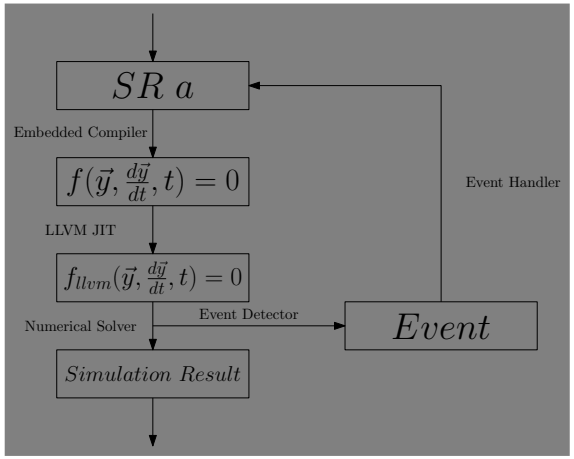
Our current FHM instance is called ***Hydra***:

- Embedding in Haskell using ***quasiquoting***.
- Model ***transformed*** to form suitable for simulation by an embedded compiler.
- The resulting system function and event detection functions are ***compiled to native code*** using the Low-Level Virtual Machine (LLVM) JIT compiler.
- System function and event detector passed to state-of-the art numerical ***solvers from SUNDIALS*** suite (from LLNL).

Prototype Hydra Implementation (2)

- At events:
  - Continuous integration stops.
  - Event-related information propagated back to the functional layer.
  - A new signal relation is computed.
  - The new relation is compiled into a new system function and a new event detection function.
  - Continuous integration resumes.

Prototype Hydra Implementation (3)



Prototype Hydra Implementation (4)

Status:

- Can simulate breaking pendulum and other models that undergo drastic structural changes.
- Have not yet looked in earnest at issues like state transfer across structural changes. Currently done explicitly.
- Have looked at some aspects of a suitable domain-specific type system.

Final Thoughts

- The area of modelling and simulation offers many opportunities for DSLs.
- Programming language research has a lot to offer.
- However, not always an “easy sell”:
  - Deep understanding of the domain needed.
  - Keeping things sufficiently simple can be a challenge.
  - Ideally, an inside accomplice who understands the benefits of principled, modern, language design is needed.