

Functional Hybrid Modelling

George Giorgidze Henrik Nilsson

Functional Programming Laboratory
School of Computer Science
University of Nottingham

University of Leicester
2010 Apr 28

Outline

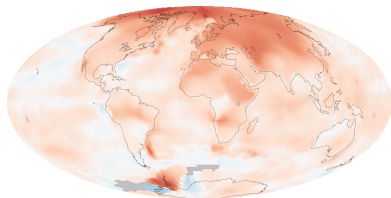
- ▶ Modelling and Simulation
- ▶ Functional Hybrid Modelling (FHM)

Modelling and Simulation

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

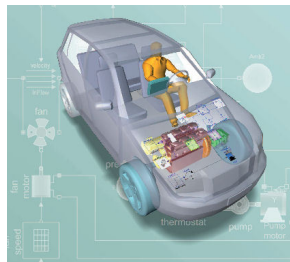
- ▶ Science

- ▶ Understanding
- ▶ Prediction



- ▶ Engineering

- ▶ Development
- ▶ Optimisation
- ▶ Safety



Computers do not have to be involved ...



But usually they are

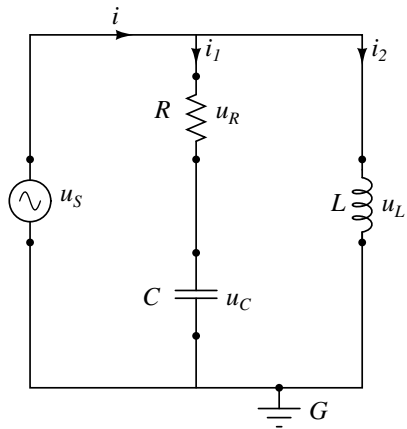
Modelling and simulation has been a main application of digital computers from the start:

- ▶ Monte Carlo simulation of nuclear detonation (Manhattan project, Los Alamos)

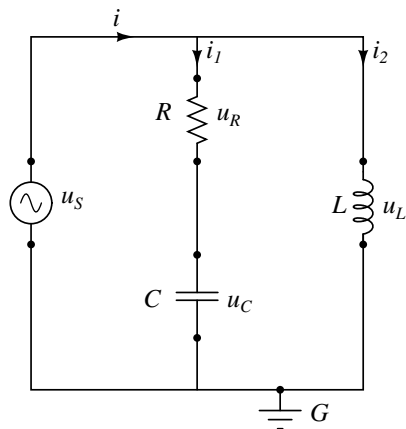
Recent examples:

- ▶ Los Alamos molecular ribosome model: 2.64 million atoms.
- ▶ The Blue Brain Project: Simulation of 10000 neurons on 8192-processor IBM Blue Gene super computer.

Simple Electrical Circuit



Simple Electrical Circuit



Differential Algebraic Equation
(DAE)

$$f\left(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t\right) = 0$$

$$u_S = \sin(2\pi t)$$

$$u_R = R \cdot i_1$$

$$C \cdot \frac{du_C}{dt} = i_1$$

$$L \cdot \frac{di_2}{dt} = u_L$$

$$i_1 + i_2 = i$$

$$u_R + u_C = u_S$$

$$u_S = u_L$$

M&S with General Purpose Programming Languages

Ordinary Differential Equation (ODE) in **explicit** and **sorted** form

$$\frac{d\vec{x}}{dt} = f(\vec{x}, \vec{u}, t)$$

$$u_S = \sin(2\pi t)$$

$$u_L = u_S$$

$$\frac{di_2}{dt} = \frac{u_L}{L}$$

$$u_R = u_S - u_C$$

$$i_1 = \frac{u_R}{R}$$

$$\frac{du_C}{dt} = \frac{i_1}{C}$$

$$i = i_1 + i_2$$

M&S with General Purpose Programming Languages

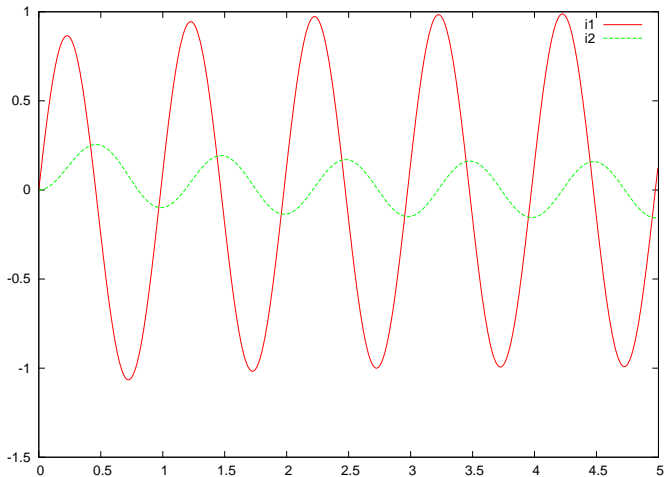
Ordinary Differential Equation (ODE) in **explicit** and **sorted** form

$$\frac{d\vec{x}}{dt} = f(\vec{x}, \vec{u}, t)$$

```
while (...) {  
    us = sin(2 * pi * t);  
    ul = us;  
    di2 = (ul / l) * dt;  
    ur = us - uc;  
    i1 = ur / r;  
    duc = (i1 / c) * dt;  
    i = i1 + i2;  
    print (t, i1, i2);  
    t = t + dt; uc = uc + duc; i2 = i2 + di2;  
}
```

$$\begin{aligned} u_S &= \sin(2\pi t) \\ u_L &= u_S \\ \frac{di_2}{dt} &= \frac{u_L}{L} \\ u_R &= u_S - u_C \\ i_1 &= \frac{u_R}{R} \\ \frac{du_C}{dt} &= \frac{i_1}{C} \\ i &= i_1 + i_2 \end{aligned}$$

Simulation Result



DSLs for Modelling and Simulation

The need for **domain-specific languages** (DSLs) 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

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)
- ▶ ...

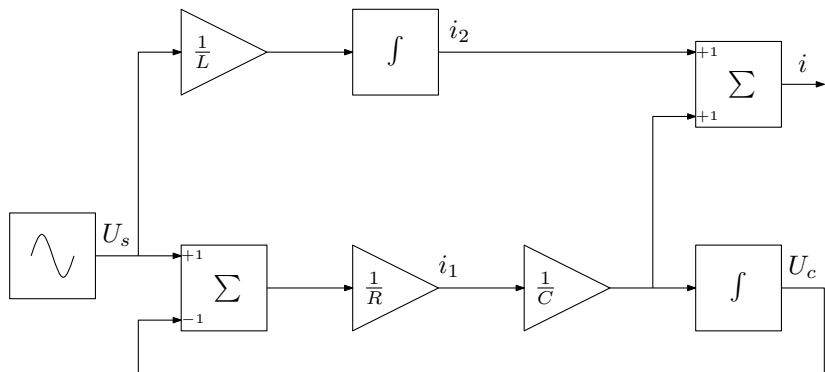
Causal Modelling

- ▶ Ordinary Differential Equation (ODE) in **explicit** form:

$$\frac{d\vec{x}}{dt} = f(\vec{x}, \vec{u}, t)$$

- ▶ **Causality** (cause-effect relationship) given by the modeller.
- ▶ Causal modelling is the dominating modelling paradigm (e.g. Simulink from MathWorks).

Simple Circuit Model in Simulink

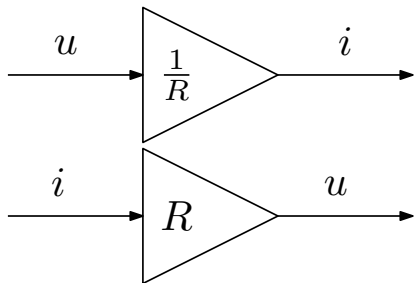


Causal Modelling

Ohm's law:

$$i = \frac{u}{R}$$

$$u = R \cdot i$$



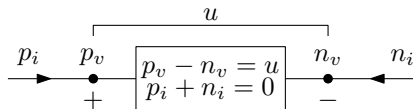
Non-Causal Modelling

- ▶ Differential Algebraic Equation (DAE) in **implicit** form:

$$f\left(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t\right) = 0$$

- ▶ Causality inferred by simulation tool from usage context.
- ▶ Non-causal modelling is a fairly recent development (e.g. Modelica, Dymola, MapleSim)

Non-causal modelling using FHM Approach



twoPin = **sigrel** $((p_i, p_v), (n_i, n_v), u)$ where

$$p_v - n_v = u$$

$$p_i + n_i = 0$$

resistor r = **sigrel** $((p_i, p_v), (n_i, n_v))$ where

$$\text{twoPin} \diamond ((p_i, p_v), (n_i, n_v), u)$$

$$r * p_i = u$$

resistor r = **sigrel** $((p_i, p_v), (n_i, n_v))$ where

$$p_v - n_v = u$$

$$p_i + n_i = 0$$

$$r * p_i = u$$

Non-causal modelling using FHM Approach

resistor $r = \mathbf{sigrel} ((p_i, p_v), (n_i, n_v))$ **where**
 $twoPin \diamond ((p_i, p_v), (n_i, n_v), u)$
 $r * p_i = u$

inductor $l = \mathbf{sigrel} ((p_i, p_v), (n_i, n_v))$ **where**
 $twoPin \diamond ((p_i, p_v), (n_i, n_v), u)$
 $l * der p_i = u$

capacitor $c = \mathbf{sigrel} ((p_i, p_v), (n_i, n_v))$ **where**
 $twoPin \diamond ((p_i, p_v), (n_i, n_v), u)$
 $c * der u = p_i$

Non-causal modelling using FHM Approach

simpleCircuit = **sigrel** (*i*, *u*) **where**

vSourceAC 1 1 $\diamond ((acp_i, acp_v), (acn_i, acn_v))$

resistor 1 $\diamond ((rp_i, rp_v), (rn_i, rn_v))$

inductor 1 $\diamond ((lp_i, lp_v), (ln_i, ln_v))$

capacitor 1 $\diamond ((cp_i, cp_v), (cn_i, cn_v))$

ground $\diamond (gp_i, gp_v)$

connect flow *acp_i rp_i lp_i*

connect *acp_v rp_v lp_v*

connect flow *rn_i cp_i*

connect *rn_v cp_v*

connect flow *acn_i cn_i ln_i gp_i*

connect *acn_v cn_v ln_v gp_v*

i = *acp_i*

u = *acp_v* - *acn_v*

Non-causal modelling using FHM Approach

simpleCircuit = **sigrel** (*i*, *u*) **where**

vSourceAC 1 1 $\diamond ((acp_i, acp_v), (acn_i, acn_v))$

resistor 1 $\diamond ((rp_i, rp_v), (rn_i, rn_v))$

inductor 1 $\diamond ((lp_i, lp_v), (ln_i, ln_v))$

capacitor 1 $\diamond ((cp_i, cp_v), (cn_i, cn_v))$

ground $\diamond (gp_i, gp_v)$

$$acp_i + rp_i + lp_i = 0$$

$$acp_v = rp_v = lp_v$$

$$rn_i + cp_i = 0$$

$$rn_v = cp_v$$

$$acn_i + cn_i + ln_i + gp_i = 0$$

$$acn_v = cn_v = ln_v = gp_v$$

$$i = acp_i$$

$$u = acp_v - acn_v$$

Functional Hybrid Modelling (FHM)

- ▶ FHM extends a **purely functional language** with a few key abstractions (e.g. **signal relation**) for supporting non-causal modelling
- ▶ FHM approach is more **declarative** and high-level, i.e. focuses on **what** to model rather than **how** to simulate
- ▶ **First class** non-causal signal relations allowing for **higher-order** and **structurally dynamic** modelling and simulation.

Higher-order Signal Relations

$serial :: SR (Pin, Pin) \rightarrow SR (Pin, Pin) \rightarrow SR (Pin, Pin)$

$serial\ sr1\ sr2 = [hydra\ |$

sigrel $((p_i, p_v), (n_i, n_v))$ **where**

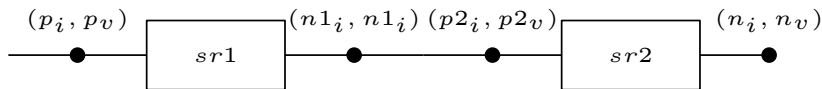
$\$ sr1 \$ \diamond ((p_i, p_v), (n1_i, n1_v))$

$\$ sr2 \$ \diamond ((p2_i, p2_v), (n_i, n_v))$

connect flow $n1_i\ p2_i$

connect $n1_v\ p2_v$

$]$



Wires

$wire :: SR (Pin, Pin)$

$wire = [hydra \mid$

sigrel $((p_i, p_v), (n_i, n_v))$ **where**

$\$ twoPin \$ \diamond((p_i, p_v), (n_i, n_v), u)$

$u = 0$

$\mid]$

Wires

$wire :: SR (Pin, Pin)$

$wire = [hydra \mid$

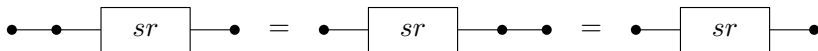
sigrel $((p_i, p_v), (n_i, n_v))$ **where**

$\$ twoPin \$ \diamond((p_i, p_v), (n_i, n_v), u)$

$u = 0$

$\mid]$

$wire \text{ 'serial' } sr = sr \text{ 'serial' } wire = sr$



Good Old Fold

serialise :: [SR (Pin, Pin)] → SR (Pin, Pin)

serialise = foldr serial wire

Good Old Fold

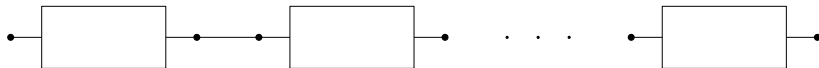
$serialise :: [SR (Pin, Pin)] \rightarrow SR (Pin, Pin)$

$serialise = foldr serial wire$

$foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

$foldr serial wire [sr1, sr2, \dots, srn] =$

$sr1 \text{ 'serial' } (sr2 \text{ 'serial' } \dots (srn \text{ 'serial' } wire))$



Parallel

$parallel :: SR (Pin, Pin) \rightarrow SR (Pin, Pin) \rightarrow SR (Pin, Pin)$

$parallel\ sr1\ sr2 = [hydra\ |$

sigrel $((p_i, p_v), (n_i, n_v))$ **where**

$\$ sr1 \$ \diamond ((p1_i, p1_v), (n1_i, n1_v))$

$\$ sr2 \$ \diamond ((p2_i, p2_v), (n2_i, n2_v))$

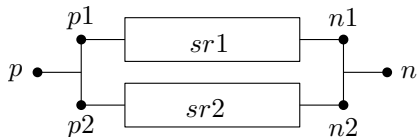
connect flow $p_i\ p1_i\ p2_i$

connect $p_v\ p1_v\ p2_v$

connect flow $n_i\ n1_i\ n2_i$

connect $n_v\ n1_v\ n2_v$

]]



No Wires

$noWire :: SR (Pin, Pin)$

$noWire = [hydra |$

sigrel $((p_i, p_v), (n_i, n_v))$ **where**

$\$ twoPin \$ \diamond((p_i, p_v), (n_i, n_v), u)$

$p_i = 0$

$]$

No Wires

$noWire :: SR (Pin, Pin)$

$noWire = [hydra |$

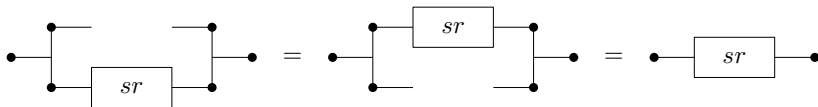
sigrel $((p_i, p_v), (n_i, n_v))$ **where**

$\$ twoPin \$ \diamond((p_i, p_v), (n_i, n_v), u)$

$p_i = 0$

$]$

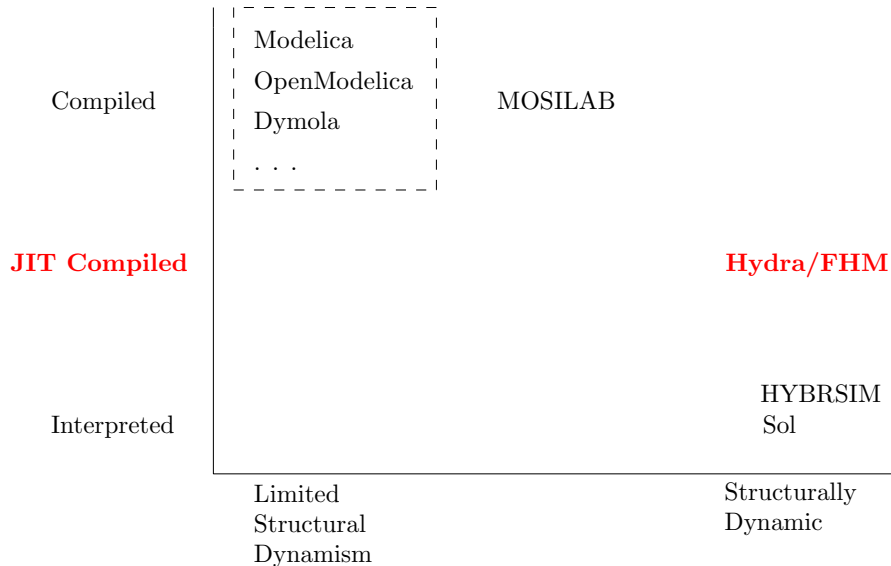
$noWire$ 'parallel' $sr = sr$ 'parallel' $noWire = sr$



Parallelise

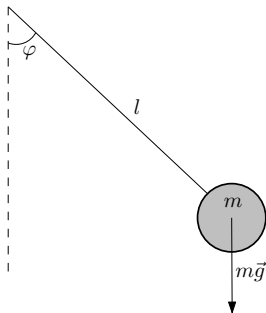
parallelise :: [SR (Pin, Pin)] → SR (Pin, Pin)
parallelise = foldr parallel noWire

Non-Causal Modelling Languages

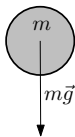


Breaking Pendulum Example

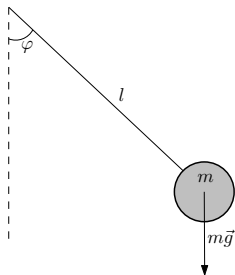
Mode 1 - Pendulum



Mode 2 - Free Fall



Pendulum Mode



type *Coordinate* = (\mathbb{R}, \mathbb{R})

type *Velocity* = (\mathbb{R}, \mathbb{R})

type *Body* = (*Coordinate*, *Velocity*)

pendulum :: $\mathbb{R} \rightarrow \mathbb{R} \rightarrow SR$ *Body*

pendulum $l \varphi_0 = [hydra |$

sigrel $((x, y), (v_x, v_y))$ **where**

init $\varphi = \$\varphi_0\$$

init $der \varphi = 0$

init $v_x = 0$

init $v_y = 0$

$x = \$l\$ * \sin \varphi$

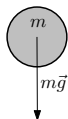
$y = - \$l\$ * \cos \varphi$

$(v_x, v_y) = (der x, der y)$

$der (der \varphi) + (\$g / l\$) * \sin \varphi = 0$

]]

Free Fall Mode



```
freeFall :: Body → SR Body
freeFall ((x0, y0), (vx0, vy0)) = [hydra |
  sigrel ((x, y), (vx, vy)) where
    init (x, y)      = ($x0$, $y0$)
    init (vx, vy)  = ($vx0$, $vy0$)
    (der x, der y)   = (vx, vy)
    (der vx, der vy) = (0, -$g$)
  ]
```

Combining the Two Modes

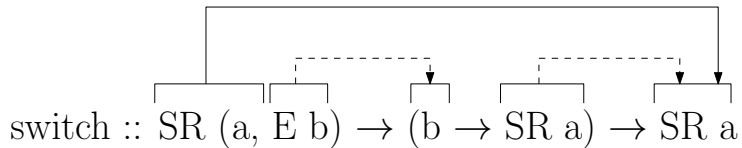
```
pendulumBE :: ℝ → ℝ → ℝ → SR (Body, E Body)
pendulumBE t l φ₀ = [hydra |
  sigrel (((x, y), (vₓ, vᵧ)), event e) where
    $ pendulum l φ₀ $ ◇ ((x, y), (vₓ, vᵧ))
  event e = ((x, y), (vₓ, vᵧ)) when time = $t $
  ]
```

```
breakingPendulum :: SR Body
```

```
breakingPendulum = switch (pendulumBE 10 1 (pi / 4)) freeFall
```

```
switch :: SR (a, E b) → (b → SR a) → SR a
```

The Switch Combinator



Embedded Domain Specific Languages (EDSLs)

Embedding is a powerful and popular way to implement **domain-specific languages** (DSLs).

There are two basic approaches to language embeddings:

- ▶ **Shallow:**
 - ▶ Domain-specific notions expressed directly in host language terms
- ▶ **Deep:**
 - ▶ Building representation of domain-specific programs as data
 - ▶ Providing interpreter or compiler

Embedded Domain Specific Languages (EDSLs)

Embedding is a powerful and popular way to implement **domain-specific languages** (DSLs).

There are two basic approaches to language embeddings:

- ▶ **Shallow:**
 - ▶ Domain-specific notions expressed directly in host language terms
- ▶ **Deep:**
 - ▶ Building representation of domain-specific programs as data
 - ▶ Providing interpreter or compiler

Mixed-level embedding combines the two approaches.

Iterative Staging and JIT Compilation

- ▶ **Compilation**: standard tool for EDSLs
 - ▶ Program generation, compilation and execution with a **fixed number of stages**
- ▶ **Iterative staging**: repeated program generation and execution
 - ▶ This work: efficient implementation approach for an iteratively staged EDSL using **mixed-level embedding** and **JIT compilation**

Internal (Untyped) Representation

```
data SigRel =  
  SigRel      Pattern [Equation]  
  | SigRelSwitch SigRel (Expr → SigRel)
```

```
data Equation =  
  EquationInit Expr Expr  
  | EquationEq Expr Expr  
  | EquationEvent String Expr Expr  
  | EquationSigRelApp SigRel Expr
```

Typed Combinators

- ▶ Phantom types
- ▶ Typing EDSL in Haskell
- ▶ Haskell type checking (GHC)

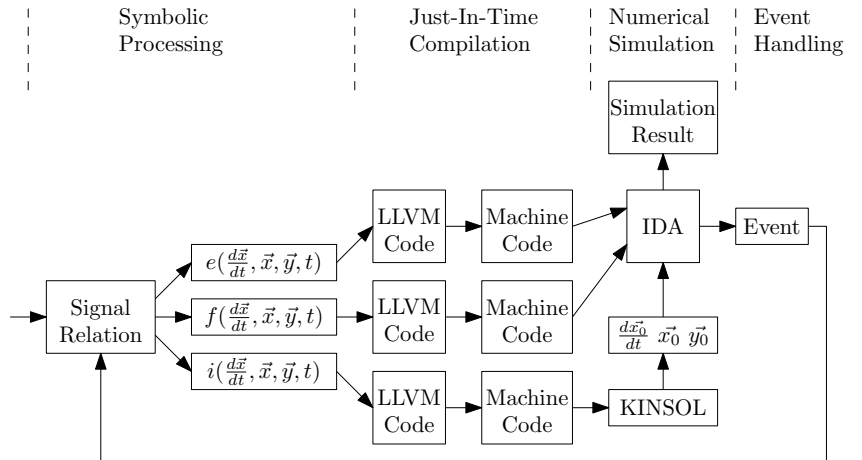
```
data SR a      = SR      SigRel  
data PatternT a = PatternT Pattern  
data ExprT a   = ExprT   Expr  
data E a
```

```
sigrel :: PatternT a → [Equation] → SR a  
sigrel (PatternT p) eqs = SR (SigRel p eqs)
```

```
switch :: SR (a, E b) → (b → SR a) → SR a
```

```
equationEq :: ExprT a → ExprT a → Equation  
equationSigRelApp :: SR a → ExprT a → Equation
```

Execution model of Hydra



Conclusions

- ▶ Novel approach to the **design** and **implementation** of non-causal modelling languages
 - ▶ Functional approach to non-causal modelling
 - ▶ Modelling systems that main-stream non-causal language can not handle
 - ▶ First compiled implementation of a non-causal modelling language that supports highly structurally dynamic systems
- ▶ EDSL approach
 - ▶ Mixed-level embedding
 - ▶ Compilation of iteratively staged EDSL
 - ▶ JIT compilation through LLVM framework
- ▶ Papers and implementation:
<http://cs.nott.ac.uk/~ggg/>

Thank You

Bonus Slides

Performance

	Pendulum $t \in [0, 10)$		Free Fall $t \in [10, 20]$	
	CPU Time		CPU Time	
	s	%	s	%
Symbolic Processing	0.0001	0.2	0.0000	0.0
JIT Compilation	0.0110	18.0	0.0077	9.1
Numerical Simulation	0.0500	81.8	0.0767	90.9
Event Handling	0.0000	0.0	-	-
Total	0.0611	100.0	0.0844	100.0

Table: Time profile of the breaking pendulum simulation

Performance

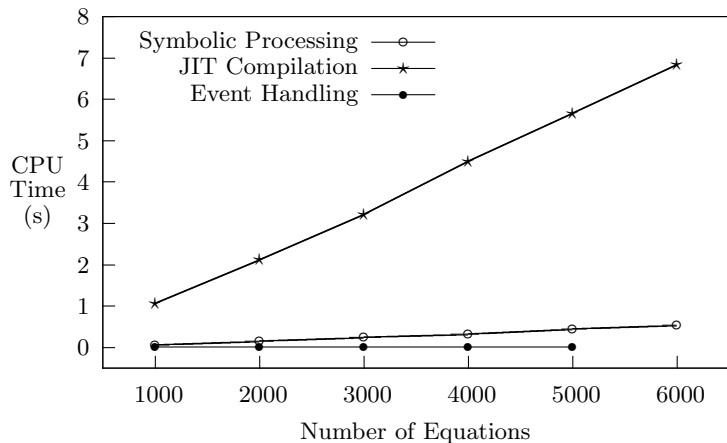


Figure: Plot demonstrating how CPU time spent on mode switches grows as number of equations increase in structurally dynamic RLC circuit simulation

Shallow

```
type Region = ( $\mathbb{R}$ ,  $\mathbb{R}$ )  $\rightarrow$  Bool  
circle ::  $\mathbb{R}$   $\rightarrow$  Region  
circle r =  $\lambda(x, y) \rightarrow \text{sqrt } (x \uparrow 2 + y \uparrow 2) \leq r$   
rectangle ::  $\mathbb{R}$   $\rightarrow$   $\mathbb{R}$   $\rightarrow$  Region  
union :: Region  $\rightarrow$  Region  $\rightarrow$  Region  
...
```

Deep

```
data Region =  
  Circle  $\mathbb{R}$   
  | Rectangle  $\mathbb{R}$   $\mathbb{R}$   
  | Union Region Region  
  ...
```

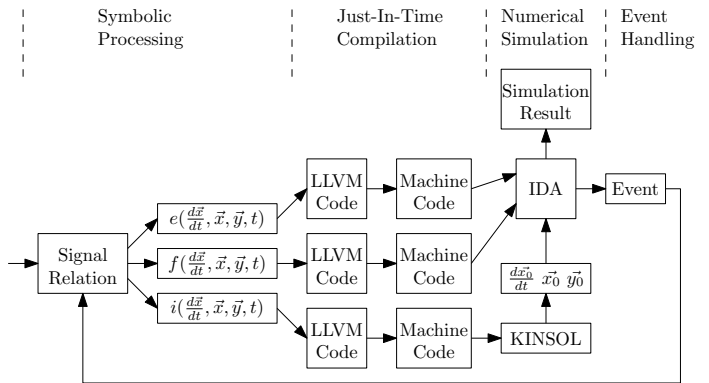
Given a parametrised signal relation:

$$sr1 :: \mathbb{R} \rightarrow SR ((\mathbb{R}, \mathbb{R}), E \mathbb{R})$$

we can recursively define a signal relation sr that describes an overall behaviour by “stringing together” the behaviours described by $sr1$:

$$sr :: \mathbb{R} \rightarrow SR (\mathbb{R}, \mathbb{R})$$
$$sr \ x = switch (sr1 \ x) \ sr$$

Future Work



- ▶ Code reuse
- ▶ Mode switching overhead reduction
- ▶ (Soft) real-time simulation

Related Work

- ▶ **Flask**: EDSL for programming sensor networks [Mainland et al.]
- ▶ **Accelerate**: EDSL for data-parallel array computations on GPUs [Chakravarty et al.]
- ▶ **Functional Reactive Programming** (FRP), particularly Yampa [Nilsson et al.]
- ▶ DSLs embedded in **multi-stage** host language (e.g. MetaOCaml) [Taha et al.]
- ▶ Modelling and simulation languages:
 - ▶ MKL [Broman et al.]
 - ▶ MOSILAB [Nytsch-Geusen et al.]
 - ▶ Sol [Zimmer et al.]