Modelyze: Embedding Equation-Based DSLs

SYNCHRON'13

Dagstuhl, Germany, November 20, 2013

David Broman
broman@eecs.berkeley.edu

EECS Department
UC Berkeley, USA

and

Linköping University, Sweden

Modelyze contributors

David Broman
Jeremey G. Siek
Hokeun Kim

Agenda

Part I

Modelyze Overview

Part II

Formal Semantics

$$\frac{\Gamma \vdash_L e_1 \rightsquigarrow e_1' : \langle\tau_{11} \rightarrow \tau_{12}\rangle \quad \Gamma \vdash_L e_2 \rightsquigarrow e_2' : \tau_2 \quad \lceil e_2' : \tau_2 \rceil = e_2'' \quad \langle\tau_{11}\rangle \sim \lceil \tau_2 \rceil}{\Gamma \vdash_L e_1\, e_2 \rightsquigarrow e_1' @ e_2'' : \langle\tau_{12}\rangle} \text{(L-APP5)}$$

Part III

Modelyze Demo

Part II

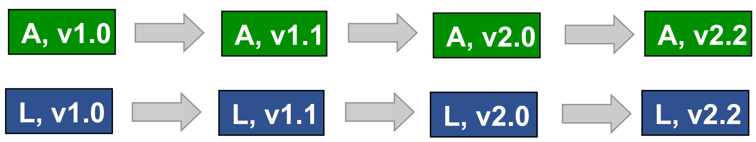
Modelyze Overview

Problem: Expressiveness and Analyzability

Cannot express all modeling or analysis needs.
Limited to what the modeling language can provide.

Language versions:

Standard library versions:



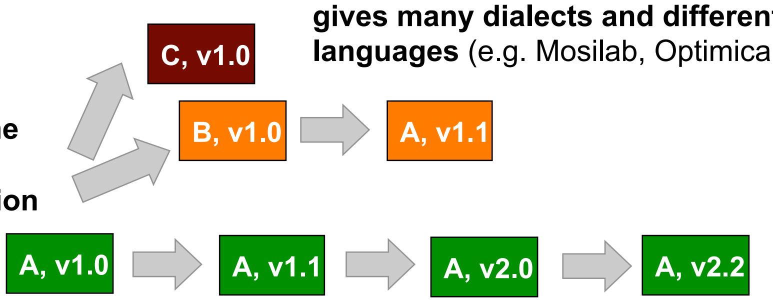
Modelica: A new language definition approximately every second year

Uses

- **Simulation**
- **Optimization**
- **Code generation for real-time**
- **Model export**
- **Grey-box system identification**

etc.

gives many dialects and different languages (e.g. Mosilab, Optimica)



What is Modelyze?

broman@eecs.berkeley.edu

Modelyze
(MODEL and analYZE)

- Purpose: Research language – addresses the expressiveness and analyzability problem by making the language <u>extensible</u>
- Small, simple, host language for embedding domain-specific languages (DSL) of different models of computation (MoC)
- Key aspect: Both the DSL and models in the DSL are defined in Modelyze
- Gradually typed functional language (call-by-value)
- Novelty: Typed symbolic expressions
- Formal semantics for a core of the language. Proven type soundness for the core.
- Prototype implementation (interpreter). Evaluated for series of equation-based DSLs.

Experimental DSLs

broman@eecs.berkeley.edu

Extensible DSLs for physical modeling

Differential-Algebraic Equations (DAE) → ModelyzeDAE

ModelyzeDAE → ModelyzeEOO

ModelyzeEOO ← Acausal connections (Electrical and Mechanical domain)

ModelyzeDAE → ModelyzeHC

ModelyzeEOO → ModelyzeHEOO

HybridCharts (DAE with modes) → ModelyzeHC

ModelyzeHC → ModelyzeHEOO

ModelyzeHEOO ← EOO + HC = HEOO

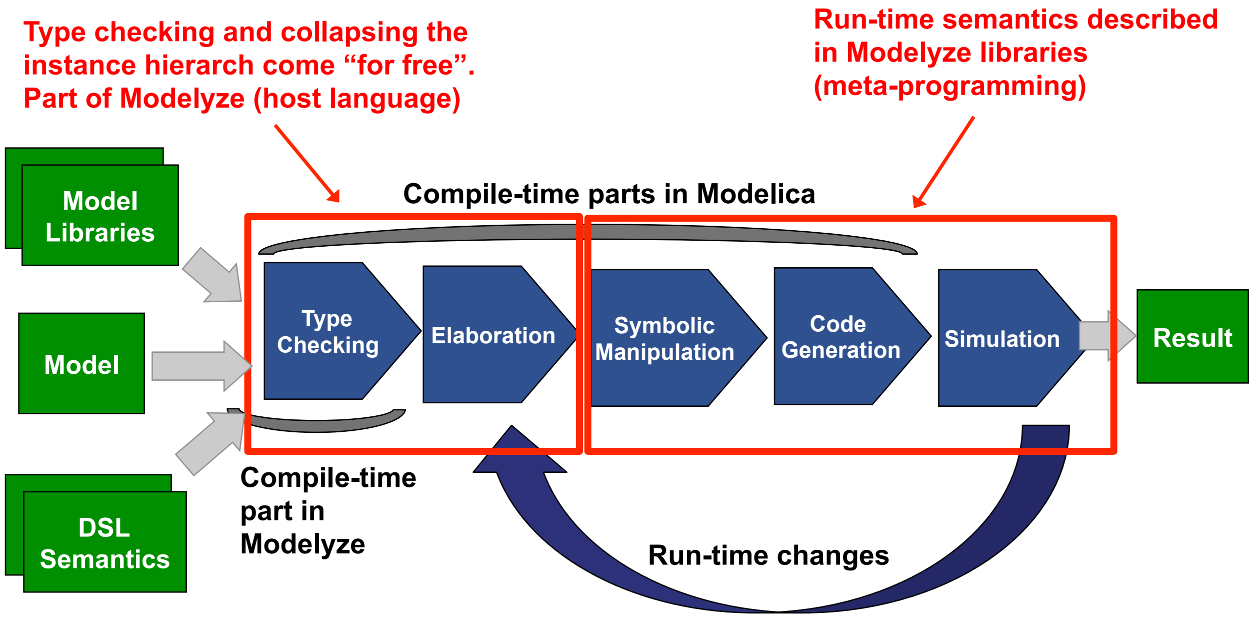
Ongoing work on combining heterogeneous DSLs for CPS

ModelyzeMA ← Master algorithm according to formalized FMI interface (see Broman et al. EMSOFT'13)

ModelyzeHEOO ModelyzeSync ModelyzeDE

Overview of the Compilation and Simulation Process

broman@eecs.berkeley.edu



Related Work

broman@eecs.berkeley.edu

Implementing DSLs

Compiler construction

- JastAdd (Ekman & Hedin, 2007)
- MetaModelica (Pop & Fritzson, 2006)

Preprocessing and template metaprogramming

- C++ Templates (Veldhuizen, 1995)
- Template Haskell (Sheard & Peyton Jones, 2002)
- Stratego/XP (Bravenboer et al., 2008)

Embedded DSLs

- Haskell DSELs, e.g., Fran (Ellito & Hudak, 1997), Lava (Bjesse et al. 1998), and Paradise(Augustsson, 2008)
- FHM (Nilsson et al., 2003)
- ForSyDe (Sander & Jantsch, 2004)
- Pure embedding (Higher-order functions, polymorphism, lazy evaluation, type classes) (Hudak, 1998)

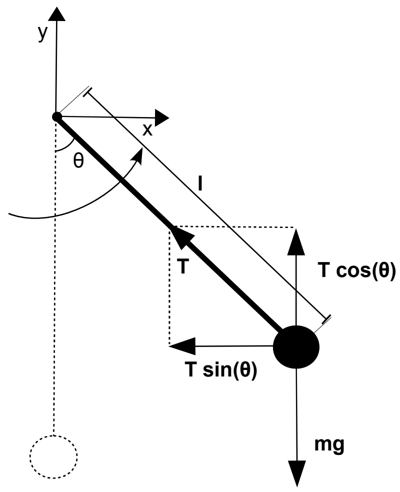
Combining Dynamic and Static Typing

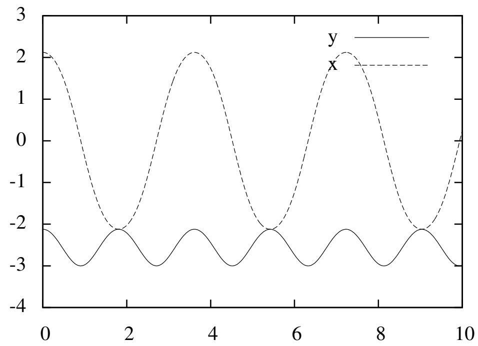
- Gradual Typing (Siek & Taha, 2007)
- Soft Typing (Cartwright & Fagan, 1991)
- Dynamic type with typecase (Abadi et al., 1991)
- Typed Scheme, Racket (Tobin-Hochstadt, Felleisen, 2008)
- Thorn, like types (Wrigstad et al., 2010)

Representing Code and Data type

- Dynamic languages LISP, Mathematica
- MetaML <T> (Taha & Sheard, 2000)
- GADT (Peyton Jones et al.,2006; Xi et al., 2003; Cheney & Ralf, 2003)
- Open Data types (Löh & Hinze, 2006)
- Pattern Calculus (Jay, 2009)
- Syntactic library (Axelsson, 2012)

Pendulum Example





$$-T \cdot \frac{x}{l} = m\ddot{x}$$

$$-T \cdot \frac{y}{l} - mg = m\ddot{y}$$

$$x^2 + y^2 = l^2$$

$$x(0) = l\sin(\theta_s)$$

$$y(0) = -l\cos(\theta_s)$$

Differential-Algebraic equations

Algebraic constraint

Initial values

Declarative Mathematical Model

Using function abstraction to define the model

Unknowns are given types but not bound to values

```
def Pendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y (-l*cos(angle));

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

Equations and initial values are defined declaratively, just as the mathematical equations

$$-T \cdot \frac{x}{l} = m\ddot{x}$$

$$-T \cdot \frac{y}{l} - mg = m\ddot{y}$$

$$x^2 + y^2 = l^2$$

$$x(0) = l\sin(\theta_s)$$

$$y(0) = -l\cos(\theta_s)$$

Declarative Mathematical Model

Which parts are part of the host language (Modelyze)?

Unknowns are internally represented as <u>typed symbols</u>

```
def Pendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y (-l*cos(angle));

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

$s : \tau$

Fresh (unique) symbol

Tagged with a type

Variable **x** is bound to fresh a symbol of type **<Real>**

$<\tau>$

Symbolic type

Release the user from annotation burden

Symbols cannot be bound to values, so **x^2** would crash at runtime

```
def Pendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y (-l*cos(angle));

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

Use quasi-quoting to mix symbolic expressions and program code?

Using MetaML syntax **< >** for quotation and **~** for anti-quoting (escape)

```
<~x^2. + ~y^2. = ~((fun t -> <t>)l^2.)>;
```

Heavy annotation burden for the end-user

Symbol Lifting Analysis (SLA)

broman@eecs.berkeley.edu

Symbol Lifting Analysis (SLA): During type checking, lift expressions that cannot be safely evaluated at runtime into symbolic expressions (data).

$$\Gamma \vdash_L e \rightsquigarrow e' : \tau$$

```
def Pendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y (-l*cos(angle));

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

Rewritten to prefix curried form

`(((/) x) l)`

where

`(/):Real-> Real -> Real`
`x:<Real>`
`l:Real`

`(((lift(/):Real-Real->Real) @ x) @ (lift l:Real))`

Resulting type `<Real>`

Division cannot be performed, lift expression to type `<Real-> Real -> Real>`.

Term `lift e:T` **wrapps** `e` **and results in type** `<T>`

Term `e1@e2` **is a symbolic application, represented as a tuple.**

Pattern Matching on Symbolic Expressions

broman@eecs.berkeley.edu

Dynamic symbolic type `<?>`

Accumulator Sets of symbolic type `<Real>`

Query for all unknowns in a model instance

```
def getUnknowns(exp:<?>, acc:(Set <Real>)) -> (Set <Real>) = {
    match exp with
    | e1 e2 -> getUnknowns(e2,getUnknowns(e1,acc))
    | sym:Real -> Set.add exp acc
    | _ -> acc
}
```

Uniform data structure, no boilplate code (matching on symbolic applications)

Match all symbols of type `<Real>` **i.e., unknowns in the model.**

`getUnknowns(Pendulum(5,3,45*pi/180),Set.empty)`

Returns a set with 3 symbols (representing `x`, `y`, **and** `T`**).**

Static Error Checking at the DSL Level

broman@eecs.berkeley.edu

Syntactically correct model (host syntax)

Static type error instead of dynamic error during translation/pattern matching.

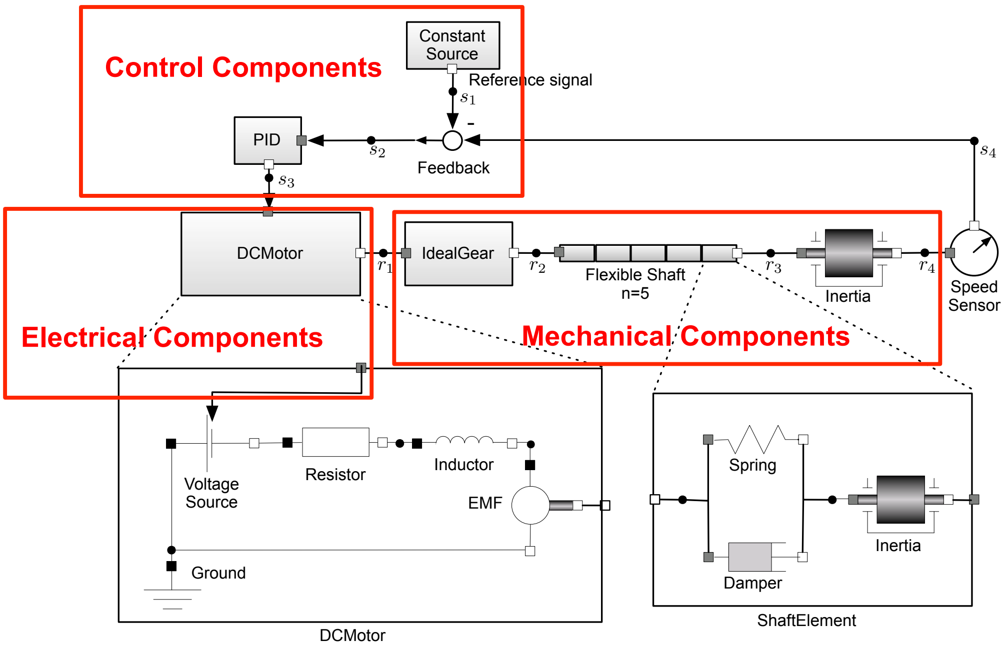
```
def ModifiedPendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y;                    //Error: Missing initial value

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

Quite intuitive error messages at the DSL level.

```
modifiedpendulum.moz 4:10-4:10 error: Missing argument
of type 'Real'.
```


Mechatronic Control Example (ModelyzeEOO)

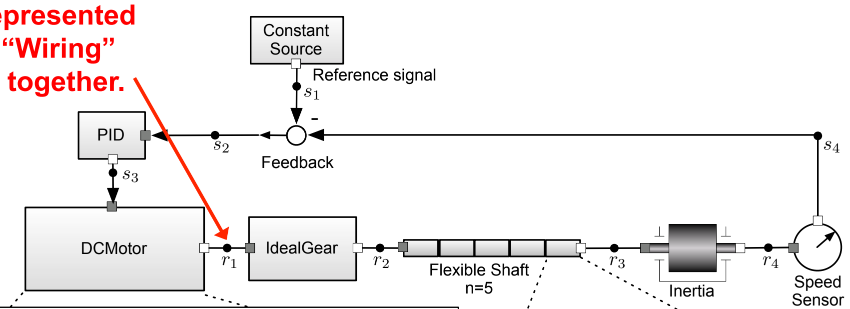
broman@eecs.berkeley.edu



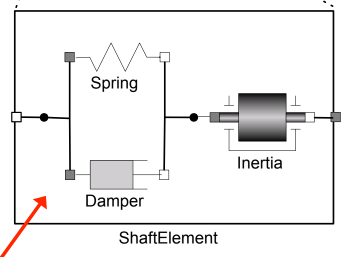
Mechatronic Control Example

broman@eecs.berkeley.edu

Nodes are represented as symbols. "Wiring" components together.




```
def CPS() = {
    def s1, s2, s3, s4:Signal;
    def r1, r2, r3, r4:Rotational;
    ConstantSource(1.0, s1);
    Feedback(s1, s4, s2);
    PID(3.0, 0.7, 0.1, 10.0, s2, s3);
    DCMotor(s3, r1);
    IdealGear(4.0, r1, r2);
    serialize(5.0, r2, r3, ShaftElement);
    Inertia(0.3, r3, r4);
    SpeedSensor(r4, s4);
}
```

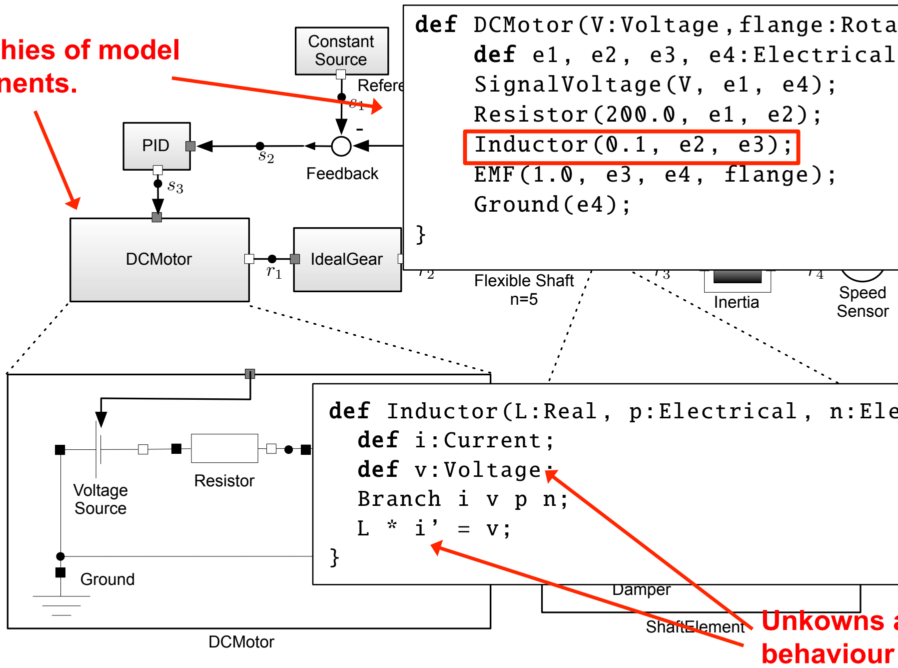


Higher-order model (higher-order function)

Mechatronic Control Example

broman@eecs.berkeley.edu

Hierarchies of model components.




```
def DCMotor(V:Voltage,flange:Rotational) = {
    def e1, e2, e3, e4:Electrical;
    SignalVoltage(V, e1, e4);
    Resistor(200.0, e1, e2);
    Inductor(0.1, e2, e3);
    EMF(1.0, e3, e4, flange);
    Ground(e4);
}
```

```
def Inductor(L:Real, p:Electrical, n:Electrical) = {
  def i:Current;
  def v:Voltage;
  Branch i v p n;
  L * i' = v;
}
```

Unkowns and behaviour equation

Part II
Formal Semantics

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \rightsquigarrow e_1' : \langle\tau_{11} \to \tau_{12}\rangle \\ \Gamma \vdash_L e_2 \rightsquigarrow e_2' : \tau_2 \\ \lceil e_2' : \tau_2 \rceil = e_2'' \\ \langle\tau_{11}\rangle \sim \lceil \tau_2 \rceil\end{array}}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e_1' @ e_2'' : \langle\tau_{12}\rangle} \text{(L-APP5)}$$

Intermediate Languages

To enable formalization and proving type soundness, we define three intermediate languages.

Both are gradually typed (Mixing static and dynamic typing).

$\lambda^{<?>}$ → $\lambda_L^{<?>}$ → $\lambda_{LC}^{<?>}$

Core — **Lifted** — **Cast inserted**

Translation step from *surface language* to core language. Includes: parsing, syntatic transformation, pattern compilation etc.

Translating from the core to Lifted language. Includes: *symbolic lifting analysis (SLA).*

Dynamic aspects made explicit by *cast insertion*. Also vital for proving type safety.

Abstract Syntax

broman@eecs.berkeley.edu

```
type Equations
def (=) : Real -> Real -> Equations
```

Core

$\lambda^{<?>}$

Ranges over ground types (Int, Real, etc.).

Function and dynamic types.

Symbolic type.

Symbolic data type (e.g., the equation above).

Ground Types	$\gamma \in \mathbb{G}$		
Symbolic Data Types	$D \in \mathbb{D}$		
Types	τ	::=	$\gamma \mid \tau \to \tau \mid ? \mid <\tau> \mid D$
Variables	$x, y \in \mathbb{X}$		
Symbols	$s \in \mathbb{S}$		
Constants	$c \in \mathbb{C}$		
Expressions	e	::=	$x \mid \lambda x{:}\tau.e \mid e\ e \mid c \mid \texttt{error} \mid$
			$\nu(\tau) \mid \texttt{case}(e, p, e, e)$
Patterns	p	::=	$\texttt{sym}{:}\tau \mid x \,@\, x \mid \texttt{lift}\ x{:}\tau$

Standard expressions (var, lambda, application, constant, error).

Case expression for eliminating symbolic data. Three forms of patterns.

"new" creates a new fresh symbol of type tau.

Type Soundness

broman@eecs.berkeley.edu

Proposition 3 (Symbolic Lifting Preserves Types). *If* $\Gamma \vdash_L e \rightsquigarrow e' : \tau$ *then* e' *is well typed in* Γ *at type* τ.

Proposition 4 (Cast Insertion Preserves Types). *If* $\Gamma \vdash_C e \rightsquigarrow e' : \tau$ *then* $\Gamma \vdash e' : \tau$.

Lemma 3 (Progress). *If* $\vdash e : \tau$ *then* $e \in$ *Values, or for all* S *there exists* S' *and* e' *such that* $e \mid S \longrightarrow e' \mid S'$, *or* $e = \texttt{error}$.

Lemma 7 (Preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \mid S \longrightarrow e' \mid S'$ *then* $\Gamma \vdash e' : \tau$.

Part III
Modelyze Demo

Conclusions

Modelyze

- Equation-based extensible modeling using an *embedded DSL approach.*
- *Symbol lifting analysis (SLA)* used to release the annotation burden from the end user
- *Gradually typed symbols* used for pattern matching and DSL-level error reporting

Thanks for listening!

See journal preprint:

David Broman and Jeremy G. Siek. **Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages**", Preprint, Submitted to Science of Computer Programming. Available as Tech. Report UCB/EECS-2012-173, University of California, Berkeley, June, 2012.

Open source implementation: http://www.eecs.berkeley.edu/~broman/