Extensible Modeling Languages and Precision Timed Infrastructures for Cyber-Physical Systems

RAWFP Workshop, Gothenburg, Sweden
May 28, 2012

David Broman
broman@eecs.berkeley.edu
EECS Department
UC Berkeley, USA
and
Linköping University, Sweden

Key collaborator related to the Modelyze project
Jeremey G. Siek

Key contributors and collaborators related to the PRET project
Edward A. Lee Isaac Liu Jan Reineke Jia Zou
Steven A. Edwards Slobadon Matic Sanjit Seshia Sungjun Kim
Jeff Jensen Hiren Patel Michael Zimmer

Agenda

Part I
Cyber-Physical Systems

Part II
Modelyze - a host language for equation-based languages

Part III
Precision Timed Infrastructure – making time an engineering abstraction
Part I
Cyber-Physical Systems

Complex Cyber-Physical Systems (CPSs)

Industrial Robots
Power Plants
Aircraft
Modeling, Simulating, and Running Cyber-Physical Systems

Model: Equation-based model

Physical system available?

System

Cyber system: Computation (embedded) + Networking

Part I
Cyber-Physical Systems

Part II
Modelyze - a host language for equation-based languages

Part III
Precision Timed Infrastructure - making time an engineering abstraction

Modeling

Compilation

Simulation with timing properties

Hardware-in-the-loop (HIL) simulation

Physical system (the plant)

Equation-based model

J_1 = M_1 - M_2
J_2 = M_0 - M_2
ω_1 = -r_2
M_1 = -r_1 M_2

Simulation with timing properties

High level modeling/programming lang. E.g., PtoMely II, Lustre/Esterel, Simulink

Part III
Precision Timed Infrastructure - making time an engineering abstraction

Timing Problem

Precision Timed Infrastructure

Extensibility Problem

Functional Embedded DSL

High level modeling/programming lang. E.g., PtoMely II, Lustre/Esterel, Simulink

Physical system (the plant)

Cyber system: Computation (embedded) + Networking

Part I
Cyber-Physical Systems

Part II
Modelyze - a host language for equation-based languages

Part III
Precision Timed Infrastructure - making time an engineering abstraction
Part II
Modelyze
a host language for equation-based languages

In collaboration with
Jeremy G. Siek
University of Colorado at Boulder

Problem: Expressiveness and Extensibility

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

Language versions:  
A, v1.0  →  A, v1.1  →  A, v2.0  →  A, v2.2

Standard library versions:  
L, v1.0  →  L, v1.1  →  L, v2.0  →  L, v2.2

Modelica: A new language definition approximately every second year
What is Modelyze?

Modelyze (MODEL and analYZE)

- Purpose: Research language – address the extensibility problem using embedded DSLs
- Small, simple, and extensible language
- Designed for embedding equation-based DSLs
- Gradually typed functional language (call-by-value)
- Typed symbolic expressions
- Formal semantics for a core of the language
- Prototype implementation (interpreter).
- Evaluated for series of equation-based DSLs.

Process Overview

- Expert in domain (e.g., Mechanics) and algorithms. Little knowledge of FP and compilers.
- External numerical libraries, e.g. SUNDIALS
- Equations generation, symbolic differentiation, index reduction etc.
- Express equations and compositions declaratively

Part I: Cyber-Physical Systems
Part II: Modelyze - a host language for equation-based languages
Part III: Precision Timed Infrastructure - making time an engineering abstraction
Challenges and Contributions

1. Provide seamless integration between host language and embedded DSL
   - Performed together with type checking.
   - Symbol Lifting Analysis

2. Make it easy to transform and analyze equations (domain expert)
   - Pattern matching on symbolic expressions
   - Gradual Typing + Typed Symbolic Expressions

3. Provide domain specific errors (model engineer)
   - Static type checking

Process Overview

Dynamic typing of symbolic elimination rules, Optional for introduction.

Domain Expert
  Creates
  Modelyze
  Domain-Specific Embedded Languages

Tradeoff between simplicity to learn, safety, and expressiveness

Domain-Specific Embedded Languages
  Models of Cyber-Physical Systems
  Creates
  Model Engineer

Static typing (explicit) for symbolic introduction rules

Part I
Cyber-Physical Systems

Part II
Modelyze - a host language for equation-based languages

Part III
Precision Timed Infrastructure - making time an engineering abstraction
Related Work

**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**
- Haskel DSELs, e.g., Fran (Ellito & Hudak, 1997), Lava (Bjesse et al. 1998), and Paradise (Augustsson, 2008)
- FHM (Nilsson et al., 2003)
- 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)

---

**Part I**

Cyber-Physical Systems

---

**Part II**

Modelyze - a host language for equation-based languages

---

**Part III**

Precision Timed Infrastructure - making time an engineering abstraction

---

**Pendulum Example**

![Diagram of a simple pendulum and its equations](image)

- **Differential-Algebraic equations**
  - \[-T \cdot \frac{x}{l} = mx\]
  - \[-T \cdot \frac{y}{l} = mg - m\dot{y}\]
  - \[x^2 + y^2 = l^2\]

- **Initial values**
  - \[x(0) = l\sin(\theta_s)\]
  - \[y(0) = -l\cos(\theta_s)\]
### Declarative Mathematical Model

**Using function abstraction to define the model**

**Unknowns are given types but not bound to values**

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

#### 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

\[
\begin{align*}
-T \cdot \frac{x}{l} &= m \ddot{x} \\
-T \cdot \frac{y}{l} - mg &= m \ddot{y} \\
x^2 + y^2 &= l^2
\end{align*}
\]

---

**Part I** Cyber-Physical Systems

**Part II** Modelyze - a host language for equation-based languages

**Part III** Precision Timed Infrastructure - making time an engineering abstraction

---

**Declarative Mathematical Model**

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

#### 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;
}

#### Symbolic type

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

---

**Part I** Cyber-Physical Systems

**Part II** Modelyze - a host language for equation-based languages

**Part III** Precision Timed Infrastructure - making time an engineering abstraction
Symbol Lifting Analysis (SLA): During type checking, lift expressions that cannot be safely evaluated at runtime into symbolic expressions (data).

\[ \Gamma \vdash e : e' : \tau \]

Rewritten to prefix curried form

\[ ((/ \ x) \ l) \]

where

\[ (/) : \text{Real} \to \text{Real} \to \text{Real} \]
\[ x : \text{Real} \]
\[ l : \text{Real} \]

\[ (((\text{lift} (/)) : \text{Real} \to \text{Real} \to \text{Real}) \ @ \ x) \ @ (\text{lift} \ l : \text{Real})) \]

Resulting type

\[ <\text{Real}> \]

Division cannot be performed, lift expression to type \[ <\text{Real} \to \text{Real} \to \text{Real}>. \]

Term \[ \text{lift} \ e : \tau \] wraps \[ e \] and results in type \[ <\tau> \]

Term \[ e_1 \text{@} e_2 \] is a symbolic application, represented as a tuple.
Symbols used to construct equation systems

```
def Pendulum(m:Real,l:Real,angle:Real) -> Equations = {
  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;
}
```

**Type of Pendulum**: Real -> Real -> Real -> Equations

Data types are open with no exhaustive checking for patterns

```
type Equations

def (=) : Real -> Real -> Equations
def (;) : Equations -> Equations -> Equations

def der : Real -> Real
def (') = der
def init : Real -> Real -> Equations
```

Pattern Matching on Symbolic Expressions

```
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
}
```

Match all symbols of type <Real>

Query for all unknowns in a model instance

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

```
getUnknowns(Pendulum(5,3,45*pi/180),Set.empty)
Returns a set with 3 symbols (representing x, y, and T).
```
Syntactically correct model (host syntax)

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

```python
def ModifiedPendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y;  //Error 1: Missing initial value
    -T*der/l = m*x'';
    //Error 2: der used incorrectly
    -T*y/l = m*g = m*y'';
    x^2 + y^2 = 1^2;
}
```

Still challenging to provide meaningful error messages.
“Expected type <Real> but found <Real->Real>”

---

**Part I**
Cyber-Physical Systems

**Part II**
Modelyze - a host language for equation-based languages

**Part III**
Precision Timed Infrastructure - making time an engineering abstraction

---

**Mechatronic Control Example**

---

**Control Components**

**Electrical Components**

**Mechanical Components**

---

Part I
Cyber-Physical Systems

Part II
Modelyze - a host language for equation-based languages

Part III
Precision Timed Infrastructure - making time an engineering abstraction
Hierarchies of model components.

Nodes are represented as symbols. “Wiring” components together.

Higher-order model (higher-order function)

Branch, used for generation e.g. sum-to-zero equations.

Unkowns and behaviour equation
Conclusions

Main takeaway points (Modelyze)

A simple research language – host language for embedding equation-based DSLs

Symbol lifting analysis (SLA) is used to release the annotation burden from the user

Typed symbols with gradual typing is used for pattern matching and DSL-level error reporting

Part II

Precision Timed Infrastructure
Making Time an Engineering Abstraction

Key contributors and collaborators related to the PRET project
Edward A. Lee
Steven A. Edwards
Jeff Jensen

Isaac Liu
Slobadon Matic
Hiren Patel

Jan Reineke
Sanjit Seshia
Michael Zimmer

Jia Zou
Sungjun Kim

Part I
Cyber-Physical Systems

Part II
Modelyze - a host language for equation-based languages

Part III
Precision Timed Infrastructure - making time an engineering abstraction
A Story…

Success?

Fly-by-wire technology controlled by software.

Safety critical ➔
Rigorous validation and certification

They have to purchase and store microprocessors for at least 50 years production and maintenance…

Why?

Apparently, the software does not specify the behaviour that has been validated and certified!

Part I
Cyber-Physical Systems

Part II
Modelyze - a host language for equation-based languages

Part III
Precision Timed Infrastructure - making time an engineering abstraction

What is PRET?

Timing is not part of the software semantics

Correct execution of programs (e.g., in C, C++, C#, Java, Scala, Haskell, OCaml) has nothing to do with how long time things takes to execute.

Traditional Approach

Programming Model

Timing Dependent on the Hardware Platform

Our Objective

Programming Model

Make time an engineering abstraction within the programming model

Timing is independent of the hardware platform (within certain constraints)
What is PRET?

- PRET = PREcision-Timed
- PRET = Predictable, REpeatable Timing
- PRET = Performance with REpeatable Timing

PRET Infrastructure

- PRET Language (Language timing semantics)
- PRET Compiler (Timing aware compilation)
- PRET Machine (Computer Architecture)

What do precision, predictable, and repeatable timing mean?

Focus on cyber-physical systems with real-time constraints

<table>
<thead>
<tr>
<th>Missed deadline</th>
<th>Hard task</th>
<th>Firm task</th>
<th>Soft task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic consequence</td>
<td>Result is useless, but causes no damage</td>
<td>Result has still some utility</td>
<td></td>
</tr>
</tbody>
</table>

Predictable timing

⇒ Guarantee correctness (WCET)

Predictable timing

⇒ Enable accuracy in nano seconds

Repeatable timing

⇒ Same platform: Testability
⇒ Changing platform: Portability

Task

(execution time in clock cycles)

Processor frequency

Deadline

(measured in e.g., ns)

Time
Rethink the ISA
Timing has to be a correctness property not only a performance (quality) property

PRET Machine
- Repeatable and predictable execution time (instructions)
- Repeatable memory access time
- Timing instructions for handling missed deadline detection

Related Work

Java Optimized Processor (JOP) (Schoeberl, 2008)
ARPRET (Andalam et al., 2009)

ARMv4 ISA extended with timing constructs

Best effort (with padding)

New instruction `get time (gt)`
```
gr r1, r2 ; get time (ns)
```

New instruction `delay until (du)`
```
du r1, r2 ; delay until 500ns have elapsed
```

We do not detect missed deadlines!
Late miss detection

- Task (execution time in clock cycles)
- Deadline
- Time (measured in e.g., ns)
- Processor frequency
- Soft task
  - Result has still some utility
  - Late miss detection
- Early and immediate miss detection
- Handled using timing exceptions
- New instructions:
  - ee - Exception on Expire
  - de - Deactivate Exception
- ARMv4 ISA extended with timing constructs
- Not handled at the ISA level
- Needs static timing analysis!
We need implementations that deliver repeatable timing

The good news
Fortunately, electronics technology delivers highly reliable and precise timing

The bad news…
The chip architectures introduces highly non-deterministic behavior (e.g., using caches, pipelines etc.).

We need to rethink the microarchitecture
• Pipelining
• Memory hierarchies
• I/O (DMA, interrupts)
• Power management
• On-chip communication
• Resource sharing (e.g., in multicore)

Our Current PRET Architecture

PTARM, a soft core on Xilinx Virtex 5 FPGA
Some of the main challenges

Relating real-time to execution-time
Execution time in clock cycles – oscillators not precise enough
- IEEE1588 clock synchronization (needs WCET margin)

Compiling with execution time
(1) Estimate WCET of ASM program
(2) WCET-aware compilation to minimize WCET

Scratchpad allocation is now a compiler problem
- Optimization problem with timing constraints. Options:
  (1) For a given frequency, optimize (average case) performance
  (2) Find lowest frequency fulfilling constraints

Making time independent of the hardware
For the same ISA, different platform parameters affect timing
(Scratchpad size, clock frequency, memory latency, etc.)
- Compiler generates a certificate – checked at load time

Conclusions

Main takeaway points (PRET)

Time is a correctness factor – not just a performance (quality) factor

Today, timing behavior is a property of the realizations of a software system

The PRET approach aims at making time an engineering abstraction within the programming model

Thank you for listening!