Modeling, Simulating, and Compiling with Timing Semantics

TU Vienna
Austria
October 4, 2012

University of Salzburg
Austria
October 2, 2012

David Broman
broman@eecs.berkeley.edu

EECS Department
UC Berkeley, USA
and
Linköping University, Sweden

Modelyze project
David Broman
Jeremey G. Siek

PRET Infrastructure
David Broman
Jian Cai
Hokeun Kim
Yooseong Kim
Edward A. Lee
Aviral Shrivastava
Michael Zimmer

PRET collaborators and alumni
Steven A. Edwards
Jeff Jensen
Sungjun Kim
Isaac Liu
Slobadan Matic
Hiren Patel
Jan Reineke
Sanjit Seshia
Jia Zou

Agenda

Part I
Cyber/Physical Co-Design Challenges

Part II
Modelyze

Part III
Precision Timed Infrastructure

Part I
Cyber/Physical Co-design Challenges

Part II
Modelyze

Part III
Precision Timed Infrastructure
Part I
Cyber/Physical Co-design Challenges

Part II
Modelyze

Part III
Precision Timed Infrastructure

Cyber-Physical Systems (CPS)

Industrial Robots
Power Plants
Aircraft
Part I
Cyber/Physical Co-design Challenges

Part II
Modelyze

Part III
Precision Timed Infrastructure

Cyber/Physical Co-Design Problem

Rapid development of CPS with high confidence of correctness is a co-design problem

The design of

Physical system
(the plant)

influence each other

The design of

Cyber system:
Computation (embedded) + Networking
Cyber/Physical Co-design Challenge #1

**System**

**Challenge #1:**
Making modeling languages expressive and analyzable

---

**Physical system** (the plant) **Cyber system:** Computation (embedded) + Networking

---

**Part I**
Cyber/Physical Co-design Challenges

**Part II**
Modelyze

**Part III**
Precision Timed Infrastructure

---

Cyber/Physical Co-design Challenge #2

**Model**

Equation-based model

Various models of computation (MoC)

---

**Model fidelity problem**

"Ensuring that the model accurately imitates the real system"

---

**Part I**
Cyber/Physical Co-design Challenges

**Part II**
Modelyze

**Part III**
Precision Timed Infrastructure

---

Challenge #2:
Compile/synthesize the model’s cyber part, such that the simulated model and the behavior of the real system coincide.

The main challenge is to guarantee correct timing behavior.

---

Expression problem

“A modeling language’s capability to handle new modeling needs”

---

Analyzability problem

“A modeling language’s capability to prove or test model properties”

---

Equation-based model

\[
\begin{align*}
J_1\dot{\omega}_1 &= M_e - M_1 \\
J_2\dot{\omega}_2 &= M_b - M_2 \\
\omega_1 &= -r\omega_2 \\
M_1 &= -r^{-1}M_2
\end{align*}
\]
Part II

Modelyze

What is Modelyze?

**Modelyze**
(MODEL and analyze)

Purpose: Research language – address the expressiveness and analyzability problem by making the language extensible

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.
Finally, in Section 5.5, we introduce a symbolic lifting analysis that is inspired by the type system of MetaML. To provide a seamless integration between Modelyze and embedded DSLs, we develop a new type system based on symbolic elimination rules. More specifically, to overcome the previously described challenges, we present the following contributions:

- We formalize the core of Modelyze and prove type safety (Section 5.5).
- We introduce a series of equation-based DSLs. We implement models from several physical domains in these DSLs (Section 5.5).
- We discuss related work and in Section 5.5, we introduce a precision timed infrastructure.

I agree, but let us wait until we finish the related work section, and then add the most relevant references back to the intro. - David

The main novelty in Modelyze from a programming language perspective is the introducction of symbolic expressions, the types for symbolic expressions must be extensible.

We discuss the benefits of using a model compiler to generate symbolic elimination rules. The Modelyze approach to modeling cyber-physical systems is shown in Figure 1: The roles, processes, and artifacts associated with the Modelyze approach to modeling cyber-physical systems.

A domain expert creates models of cyber-physical systems using Modelyze. The domain-specific embedded language performs the translation of these models into symbolic expressions. The solver expert creates numerical solvers that perform solving of these symbolic expressions. The simulation results are then used to create models of cyber-physical systems.

Express equations and compositions declaratively.

Dynamic typing of symbolic elimination rules. Optional for introduction.

Tradeoff between simplicity to learn, safety, and expressiveness.

Static typing (explicit) for symbolic introduction rules.
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 Co-design Challenges

Part II
Modelyze

Part III
Precision Timed Infrastructure

Pendulum Example

\[-T \cdot \frac{x}{l} = m \ddot{x}\]
\[x(0) = l \sin(\theta_0)\]

\[-T \cdot \frac{y}{l} - mg = m \ddot{y}\]
\[y(0) = -l \cos(\theta_0)\]

\[x^2 + y^2 = l^2\]
Different shapes:

3.1. Syntax

Extensions to the core language

They mix static and dynamic typing. The dynamic aspect is made explicit through a symbolic lifting of expressions.

Expressions in the core language. The value of expressions is application and elimination of boolean values.

Let us now define the type system for expressions in Modelyze.

We use subscripts for denoting different types.

Syntax:

Variables:

There are two new kinds of expressions in Modelyze.

1. Expressions are lifted into symbolic expressions.

The reason for symbolic lifting is, as explained, eliminating the need for the premise.

Figure 3: Abstract syntax of expressions.

Example:

Assuming no air drag, we model the forces in the pendulum.

The above mathematical model (differential equations together with initial conditions)

is a simple Newton's second law of motion (Newton's first law of motion is not enough to model the pendulum).

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

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

\[
\begin{align*}
x(0) &= l \sin(\theta_0) \\
y(0) &= -l \cos(\theta_0)
\end{align*}
\]
Symbols cannot be bound to values, so $x^2$ would crash at runtime

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

Using MetaML syntax $<>$ for quotation and $\sim$ for anti-quoting (escape)

$$x^2 + y^2 = \sim((\text{fun} \; t \rightarrow \langle t \rangle)1^2)$$

Heavy annotation burden for the end-user

Symbol Lifting Analysis (SLA)

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

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

Rewritten to prefix curried form

$$(((/ \; x) \; l)$$

where

$$x : \text{Real}
\quad l : \text{Real}$$

$$(((\text{lift}(/) : \text{Real} \rightarrow \text{Real} \rightarrow \text{Real}) \; @ \; x) \; @ (\text{lift} \; l : \text{Real}))$$

Resulting type

$$\langle \text{Real} \rangle$$

Term $\text{lift} \; e : \tau$ wraps $e$ and results in type $\langle \tau \rangle$

Term $\text{e1}@\text{e2}$ is a symbolic application, represented as a tuple.
Pattern Matching on Symbolic Expressions

broman@eecs.berkeley.edu

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

Get all unknown variables of an equation system. Function for solving linear algebraic equations. One fundamental function in such a library is to define a dynamic DSL library called Co-design Challenges.

Syntactically correct unknowns. For example, the model accumulator is returned. Note how the gradual type system is used to partially pointing out the code line of the model where the error is located. In this section we motivate and describe how this can be partially accomplished by performing static guarantees. For example, preservation of types during symbolic matching.

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

Quite intuitive error messages at the DSL level.

modifypedulum.moz 4:10-4:10 error: Missing argument of type 'Real'.
Components together.

Electrical Components

Mechanical Components

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

```
def CPS() = {
    def s1, s2, s3, s4:Signal;
    def r1: r2, r3, r4:Rotational;
    ConstantSource(1, s1);
    Feedback(s1, s4, s2);
    PID(3, 0.7, 0.1, 10, s2, s3);
    DCMotor(s3, r1);
    IdealGear(4, r1, r2);
    serialize(5, r2, r3, ShaftElement);
    Inertia(0.3, r3, r4);
    SpeedSensor(r4, s4);
}
```

Higher-order model (higher-order function)
Mechatronic Control Example

Hierarchies of model components.

Conclusions

Modelyze: main takeaway points

Addressing the modeling expressiveness and analyzability problems using an embedded DSL approach.

- Both DSLs and models are defined in the same language.
- Symbol lifting analysis (SLA) is used to release the annotation burden from the end user.
- Typed symbols with gradual typing is used for pattern matching and DSL-level error reporting.

For more information, see preprint of submitted journal paper:

A Story…

Success?

Fly-by-wire technology controlled by software.

Safety critical ➔
Rigorous validation and certification

Why?

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

They have to purchase and store microprocessors for at least 50 years production and maintenance…
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 take 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 Precision Timed (PRET) Infrastructure?

PRET Infrastructure

- PRET Language (Language with timing semantics)
- PRET Compiler (Timing aware compilation)
- PRET Machine (Computer Architecture)
What do mean by precision, predictable, and repeatable timing?

Focus on cyber-physical systems with real-time constraints

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

**Predictable timing**
- Guarantee correctness (WCET)

**Task**
- (clock cycles)

**Precision of timing**
- Enable accuracy in nano seconds

**Processor frequency**

**Repeatable timing**
- Same platform: Testability
- Changing platform: Portability

**Deadline**
- (measured in e.g., ns)

Languages with timing semantics

|--------------------|--------------------------------|---------------------------------|--------------------------------|---------------------------------|----------------------------------|

<table>
<thead>
<tr>
<th>Programming Languages</th>
<th>Real-time Concurrent C (Gehani and Ramamritham, 1991)</th>
<th>PRET-C (Andalam et al., 2009)</th>
</tr>
</thead>
</table>

Assembly Languages
- The assembly languages for todays processors lack the notion of time
We need implementations that deliver deterministic timing behavior

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)

Instruction set architecture (ISA)

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)
Padding and late miss detection

New instruction `get time (gt)`

```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gt</code></td>
<td><code>r1, r2</code> ; get time (ns)</td>
</tr>
</tbody>
</table>
```

New instruction `delay until (du)`.

Padding using delay until

Late miss Detection (soft task)

Do get time again and compare

```
Task (clock cycles)
```

**Immediate miss detection**

Handled using timing exceptions

New instructions:

- `ee` - Exception on Expire
- `de` - Deactivate Exception

**Immediate miss detection**
### Early miss detection

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{gt} r1, r2 ; get time (ns)</td>
<td></td>
</tr>
<tr>
<td>\texttt{adds} r2, r2, #500 ; add 500 ns</td>
<td></td>
</tr>
<tr>
<td>\texttt{adc} r1, r1, #0</td>
<td></td>
</tr>
<tr>
<td>\texttt{mtfd} r1, r2 ; takes at most 500 ns</td>
<td></td>
</tr>
<tr>
<td>\texttt{du} r1, r2 ; takes at least 500 ns</td>
<td></td>
</tr>
</tbody>
</table>

- \texttt{mtfd} = meet the final deadline
- \texttt{gt} r1, r2 ; get time (ns)

At runtime – machine error exception.
(Critical error mode – “stop the train”)

To be “early” - Needs to be guaranteed by static analysis of program code

---

### Our Current PRET Architecture

**PTARM, a soft core on Xilinx Virtex 5 FPGA**

- **Xilinx Virtex 5, FPGA, 75 MHz**
- **Hardware thread**
  - registers
- **Scratch pad**
- **main memory**
- **I/O devices**

**Thread-interleaved Pipeline**
- 4 threads, 5 stage pipeline

**Scratchpad shared among threads**
- 1 thread cycle for load/store

**DRAM main memory, separate banks per thread**
- Load 4 thread cycle
- Store 1-2 thread cycles
  (1 thread cycle = 4 processor cycles)
Semantics gap between timed high level modeling languages and PRET ISA

Can we just compile directly down to PTARM?
Lots of redundant work...
PRETIL (work in progress)

Modeling Languages

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C extension with high-level timing constructs.</td>
<td>Can be seen both as an intermediate and programming language</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Programming Languages

- Expose timing constructs (gt, du, mtfd, ee, de)
- Expose allocation of virtual threads
- Abstractive away memory hierarchy (scratchpad)
- Hide hardware threads
- Encoding of flow facts
- Perform scratchpad allocation
- Perform WCET analysis

Assembly Languages

<table>
<thead>
<tr>
<th></th>
<th>Related work WCC (Falk and Lokuciejewski, 2010)</th>
<th>PRET ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Our approach – an optimization problem: Minimize clock frequency, or scratchpad size, such that for all tasks ( t ), WCET(( t )) \leq \text{mtfd}(t)</td>
<td></td>
</tr>
</tbody>
</table>

Conclusions

PRET Infrastructure: main takeaway points

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 construct within the programming model

For more information about PRET, see