Transferring the System Modeler code base to OCaml

Leonardo Laguna Ruiz

Wolfram MathCore

About me

- PhD in Electrical Engineering
- Interested in Synthesizers, Digital Signal Processing, mathematical modeling and compilers
- Started programming C and C++, assembly (DSPs, Z80, PIC, x86) and making programs for the HP 49G calculator

First steps in Functional Programming

Back in 2007, I needed to evaluate (in C#) string expressions like

“1/2 + 2 * a + b”

Required a lexer, parser and writing an evaluator

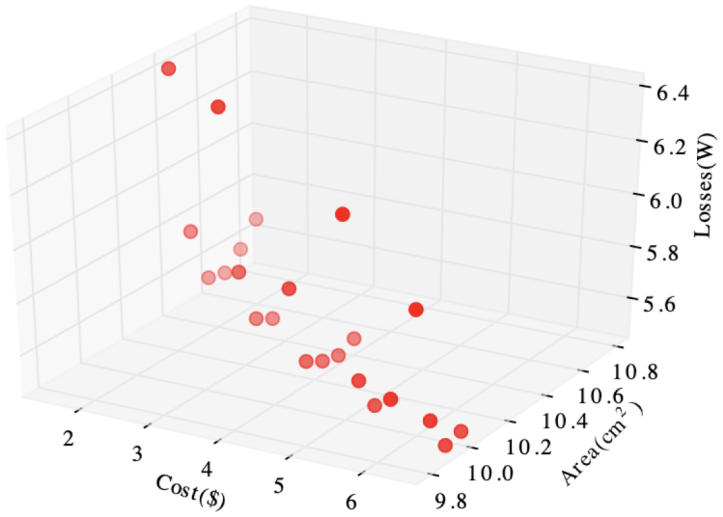
Found F# example

```
let rec eval table e =
    match e with
    | Number n -> n
    | Var s -> eval table (look s)
    | Sum (a, b) ->
      (eval table a) + (eval table b)
    | Mul (a, b) ->
      (eval table a) * (eval table b)
```

- My evaluator was less than F# 100 lines

The F# years

- Wrote F# for about 5 years
- Worked on a model compiler, simulator and optimization engine (PhD thesis)
- Learned about other languages, like Haskell, Lisp, OCaml.

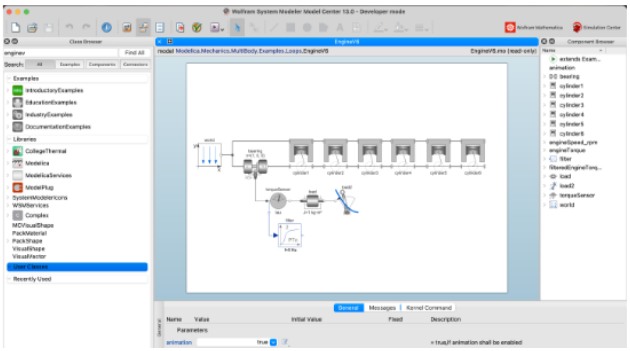


Outline

- System Modeler architecture
- Historical context of the code base
- Selecting a new language
- Moving the code base

Wolfram System Modeler

- Multi-domain modeling environment
- Uses the **Modelica** modeling language
- Integrates with the Wolfram Language

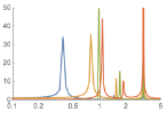


Compute $|H(i\,\omega)|$ from $Y(i\,\omega) = H(i\,\omega)\,U(i\,\omega)$:

```
idata = Table[Abs@Fourier@Table[UnitStep[t] - UnitStep[t - 2], {t, 0, 200, 0.25}], {sim, sims}];
odata = Table[
    Abs@Fourier@Table[Evaluate[sim["inertia3.w"], 1][[1]], {t, 0, 200, 0.25}], {sim, sims}];
```

Show the resonance peaks:

```
ListLogLinearPlot[odata[[All, 1 ;; 200]] / idata[[All, 1 ;; 200]],
    Joined → True, DataRange → {0, 2 Pi}, PlotRange → {{0.1, 5}, {0, 50}}]
```



Applications

- Electrical, Hydraulic, Mechanic, Thermal, Biological, Chemical, Fluid, Magnetic, etc.

Industry Examples

Aerospace & Defense

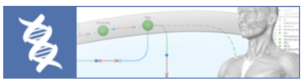
Automotive & Transportation

Heavy Equipment

Industrial Manufacturing

Consumer Products

Energy



Life Sciences

Business

Other

Personal applications

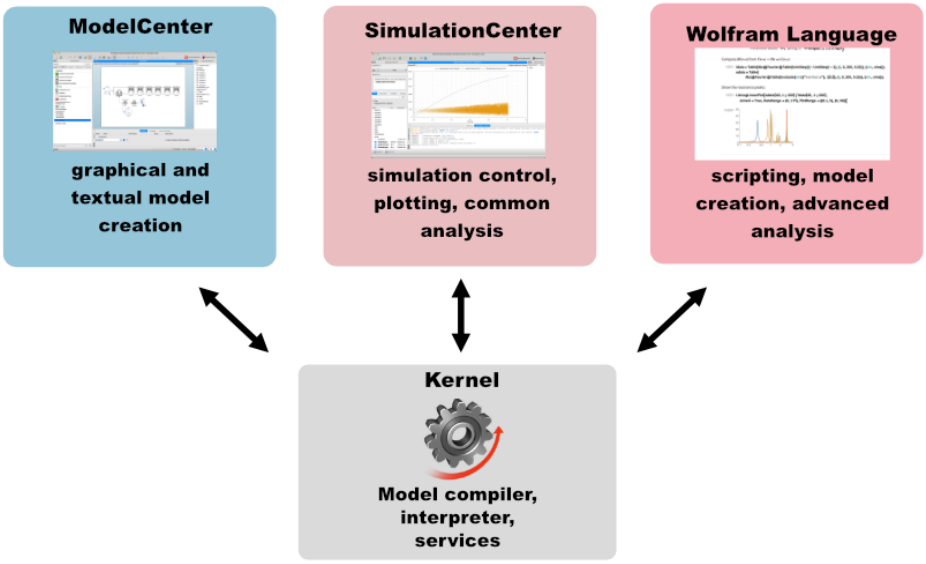
https://blog.wolfram.com/2020/07/23/digital-vintage-sound-modeling-analog-synthesizers-with-the-wolfram-language-and-system-modeler/

Digital Vintage Sound: Modeling Analog Synthesizers with the Wolfram Language and System Modeler

July 23, 2020

Leonardo Laguna Ruiz, Software Engineer, SystemModeler

Wolfram System Modeler architecture



System Modeler history

- Originally called MathModelica
- 1998 (?)MathCore Engineering founded
- 1999(?) Initial version written in Mathematica (Wolfram Language)
- 2006 MathModelica v1.0 is released
- 2007 OpenModelica starts (sister project to MathModelica)
- 2011 MathCore becomes Wolfram MathCore, Leonardo joins the company
- 2011 Wolfram SystemModeler is released

System Modeler Kernel development history

- 1995 (?) RML (Relational Meta-Language), SML-like data types
- 2005 (?) MetaModelica v1 (used RML core compiler and code generator)
- System Modeler kernel and OpenModelica kernel were written in MetaModelica

```
uniontype exp
  record NUMBER
    Real v;
  end NUMBER;
  record VAR
    String v;
  end VAR;
  record SUM
    exp e1;
    exp e2;
  end SUM;
end exp;
```

The good parts of MetaModelica

- Static types ML style
- Pattern matching
- First class functions
- Portable (generates C code)

```
function eval
  input table t;
  input exp e;
  output Real o;
algorithm
  o := matchcontinue(t, e)
    local
      exp n1,n2;
      Real v,n1v,n2v,ret;
      Integer n;
    case(_, NUMBER(v)) then v;
    case(_, SUM(n1,n2))
      equation
        n1v = eval(t, n1);
        n2v = eval(t, n2);
        ret = n1v +. n2v;
      then ret;
    case(_, VAR(s))
       equation
        ret = lookup(table, s);
       then ret;
    case(_,_)
      equation
        print("Failed to evaluate expression");
      then fail();
  end matchcontinue;
end eval;
```

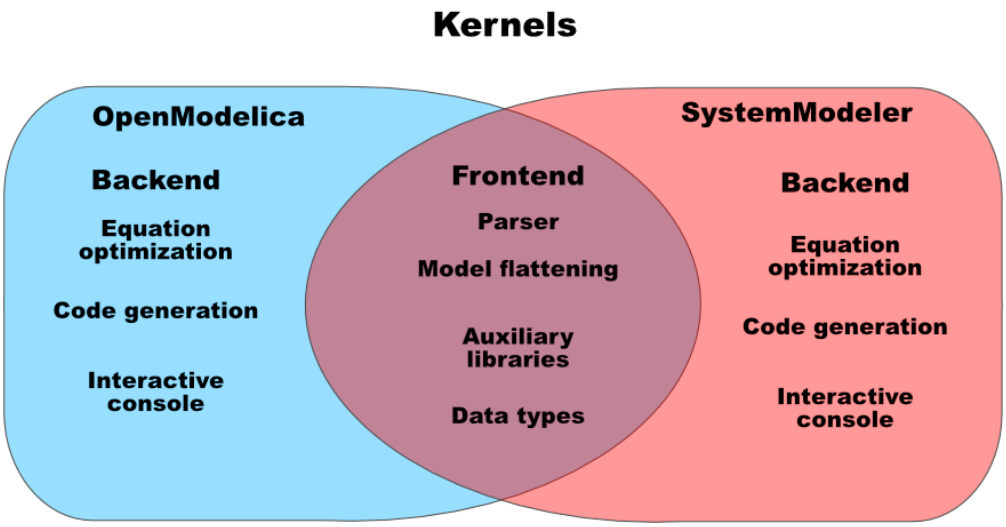
The bad parts of MetaModelica

- Verbose
- Not so good performance, slow compile speed
- No anonymous functions, no closures, no if-expressions
- No nested pattern matching, match all inputs
- Print debug only
- Difficult error handling
- Bugs: broken tail-calls, many others

```
function foo
  input   Integer i;
  output  Integer o;
algorithm
  o:=matchcontinue(i)
    local Integer a;
    case(0)
      equation
        a = canFail();
      then 0;
    case(1) then 1;
    case(_)
      equation
        print("The input is not 0 or 1");
      then fail();
  end matchcontinue;
end foo;
```

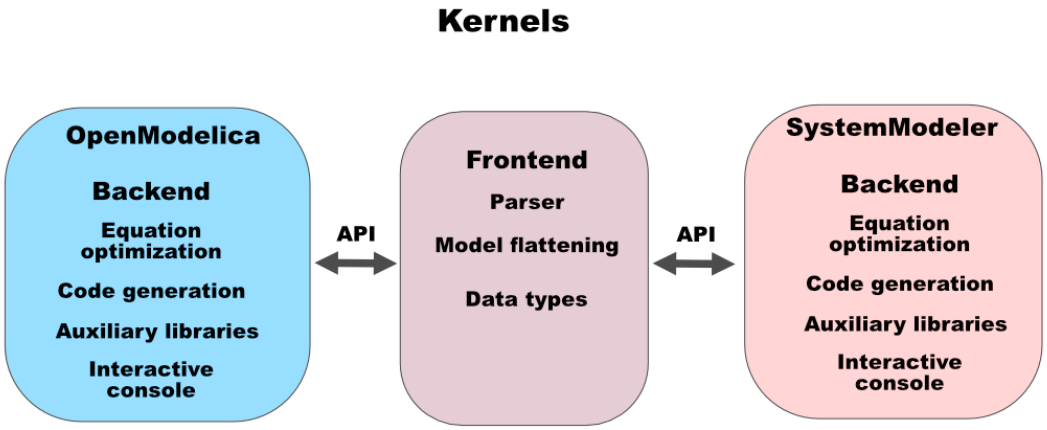
Relation between SystemModeler and OpenModelica

- Big part of the code was shared
- No stable API
- Lots of time spent upstreaming changes
- Little control on design changes or decisions



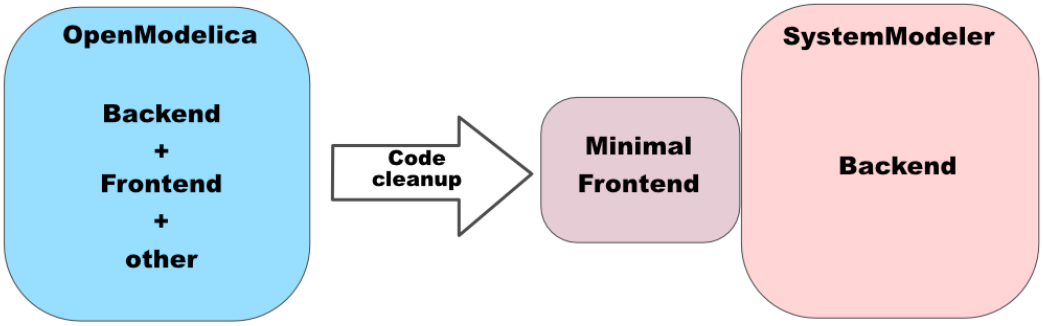
Desired scenario

- OpenModelica needed to split their project and maintain a stable API



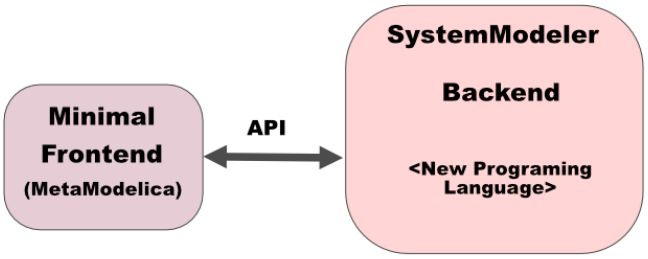
Initial approach

- F# based tool to analyze and extract the relevant parts
- Simplifies merging upstream changes
- Smaller diffs for relevant parts



Moving to other language

- The API are the data types
- The data types changed less than the functions
- The data can be communicated by serializing it



Exchanging data

- The key is automatically generating data converters

```
uniontype exp
  record NUMBER
    Real v;
  end NUMBER;
  record VAR
    String v;
  end VAR;
  record SUM
    exp e1;
    exp e2;
  end SUM;
end exp;
```

```
type exp =
  | NUMBER of float
  | VAR of string
  | SUM of exp * exp
```

```
enum Tag { NUMBER, VAR, SUM };

typedef struct {
  Tag tag;
  // NUMBER
  float n;
  // VAR
  char *s;
  // SUM
  Exp *e1;
  Exp *e2;
} Exp;
```

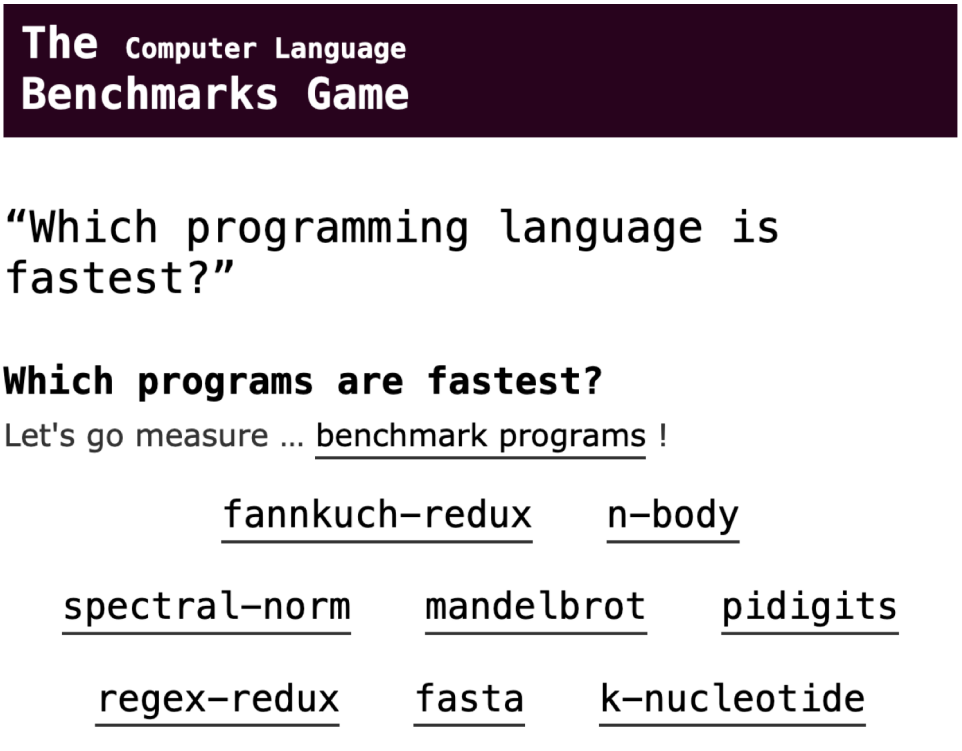
Picking a new programming language

As many as possible of the following:

- Possible to automatically convert the existing code into readable code
- Static types (catch errors at compile time)
- Union types (easier to represent our complex data)
- Pattern matching (easier to process symbolic data)
- Functional (simplifies writing complex behavior)
- Succinct (less code more meaning)
- Debugger (print-debug is not enough)
- Fast execution (good for our users)
- Fast compilation (good for us developers)

Benchmarking different languages

Existing benchmarks do not reflect our actual requirements



Our benchmark

Common tasks our kernel performs

- Perform symbolic manipulations of expressions
- Evaluate expressions
- Store and lookup values
- Common graph operations

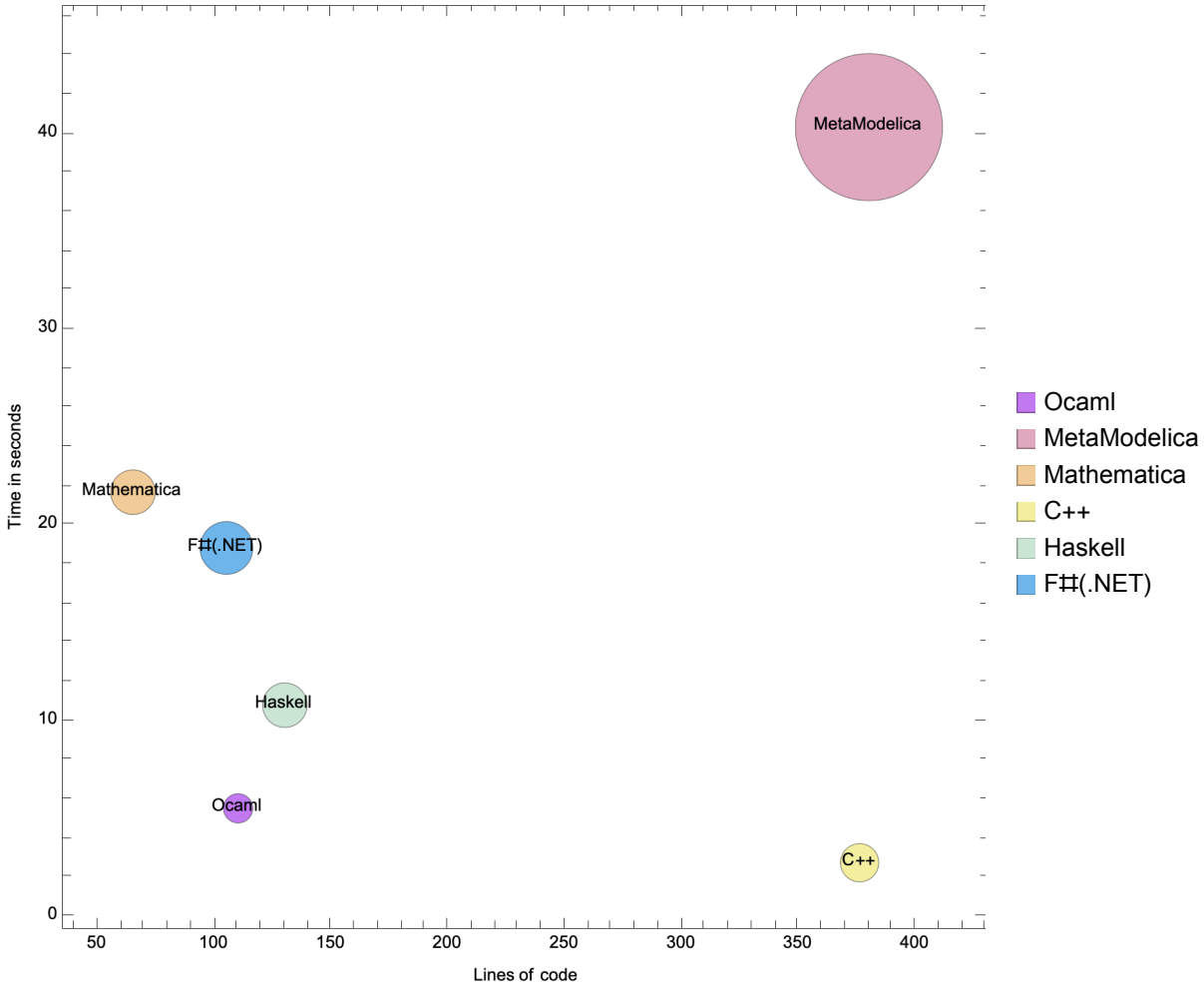
We created a small representative program

- Optimization algorithm evaluating symbolic expressions

Results

Measuring verbosity (lines of code) vs execution time.

Out[•]=

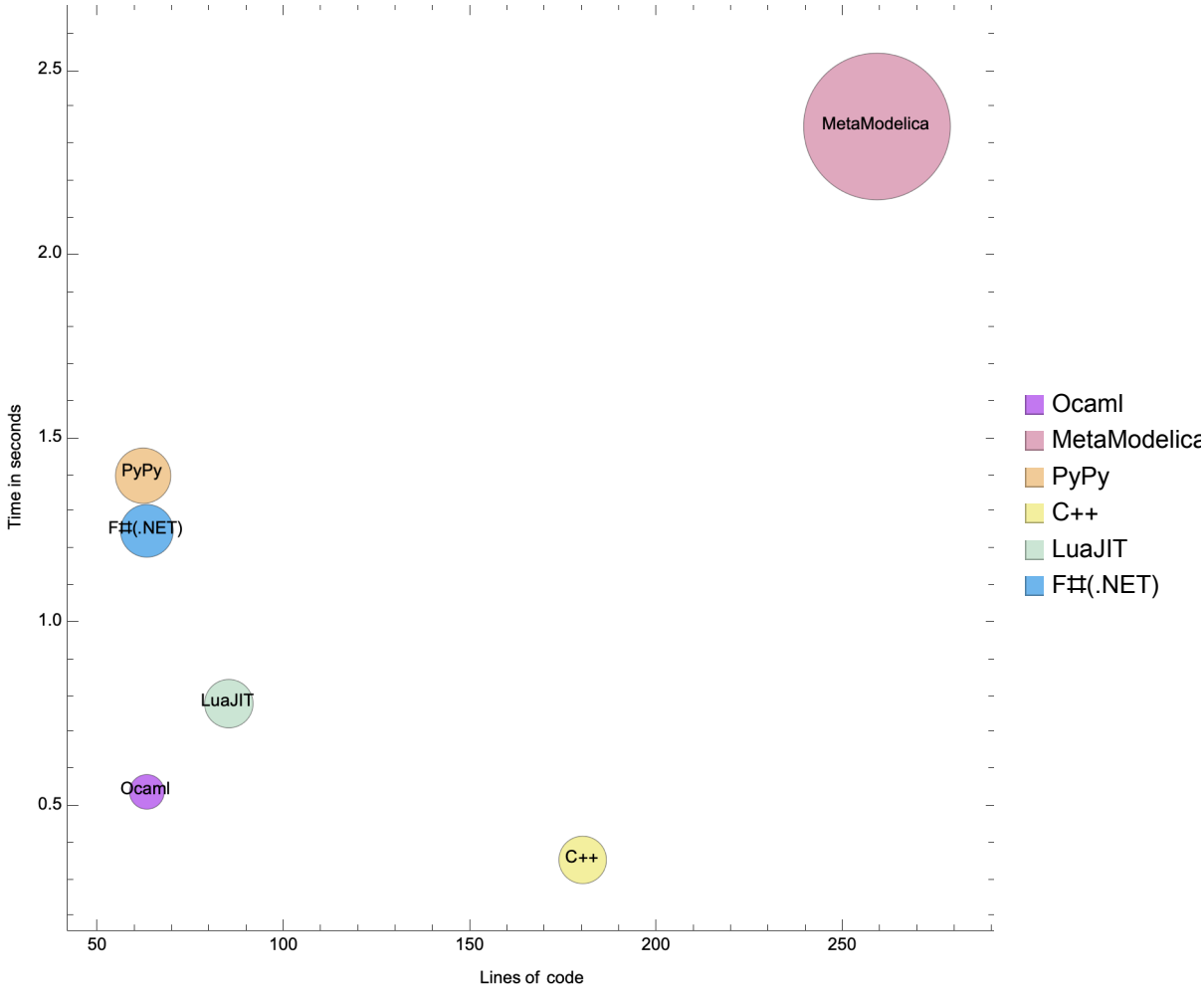


Note: this was done in 2014. No Rust, no Julia. We overlooked Java.

Second benchmark

Similar program without symbolic manipulation

Out[•]=



And the winner is...

Languages with strong cons:

- C++: automatically converted code would not be very readable probably more verbose
- F#: ran fast on Windows (.NET), but slow on Linux and Mac (using Mono as of 2014)
- Haskell: existing code has side effects (hash tables) converting the code was challenging

The real contenders

- OCaml: Fast, close to MetaModelica
- Mathematica (WL): very succinct, lost of the functionality we required is builtin. Making a good translation would be difficult.

The winner is **OCaml**

- Performance increase of ~5-10x
- Decent reduction of lines of code
- Simpler translation of code thanks to introduction of PPX syntax extensions (just when I made the benchmarks)

Cover: MetaModelica to OCaml converter

Learning OCaml while making the converter

- Parse MetaModelica code
- Analyze error paths
- Simplify the code (dead code elimination, unreachable code, match simplification)
- Simplify error paths
- Generate readable code, make it more idiomatic
- Preserve code comments

Failure propagation

```
function unsafeCall
  input  any i;
  output any o;
algorithm
  o := fail();
end unsafeCall;

function safeCall
  input  any i;
  output any o;
algorithm
  o := i +  1;
end safeCall;

function main
  output    Integer o;
  protected Integer a;
  protected Integer b;
algorithm
  a := safeCall(0);
  b := unsafeCall(0);
  o := 1;
end main;
```

```
let unsafeCall i =
    fail ()

let safeCall (i : int) =
    i + 1

let main () : int =
    let   a  = safeCall 0 in
    let%F b = unsafeCall 0 in
    return 1
```

Matchcontinue implementation

```
function foo
  input   Integer i;
  output  Integer o;
algorithm
  o:=matchcontinue(i)
    local Integer a;
    case(0)
      equation
        a = unsafeCall();
      then 0;
    case(1) then 1;
    case(_)
      equation
      then fail();
  end matchcontinue;
end foo;
```

```
let foo i =
  match%continue i with
  | 0  ->
    let%F a = unsafeCall () in
    return 0
  | 1  -> return 1
  | _  -> fail ()
```

Matchcontinue implementation

```
function foo
  input   Integer i;
  output  Integer o;
algorithm
  o:=matchcontinue(i)
    local Integer a;
    case(0)
      equation
        a = unsafeCall();
      then 0;
    case(1) then 1;
    case(_)
      equation
        print("The input is not 0 or 1");
      then fail();
  end matchcontinue;
end foo;
```

```
let foo i =
   safeReturn @@ match%continue i with
   | 0  ->
     let%F a = unsafeCall () in
     return 0
   | 1  -> return 1
   | _  -> failwith "The input is not 0 or 1"
```

Matchcontinue optimization

```
function foo
  input   Integer i;
  output  Integer o;
algorithm
  o:=matchcontinue(i)
    local Integer a;
    case(0)
      equation
        a = safeCall();
      then 0;
    case(1) then 1;
    case(_)
      equation
        print("The input is not 0 or 1");
      then fail();
  end matchcontinue;
end foo;
```

```
let foo i =
  match i with
  | 0  ->
    let a = safeCall () in
    0
  | 1  -> 1
  | _  -> failwith "The input is not 0 or 1"
```

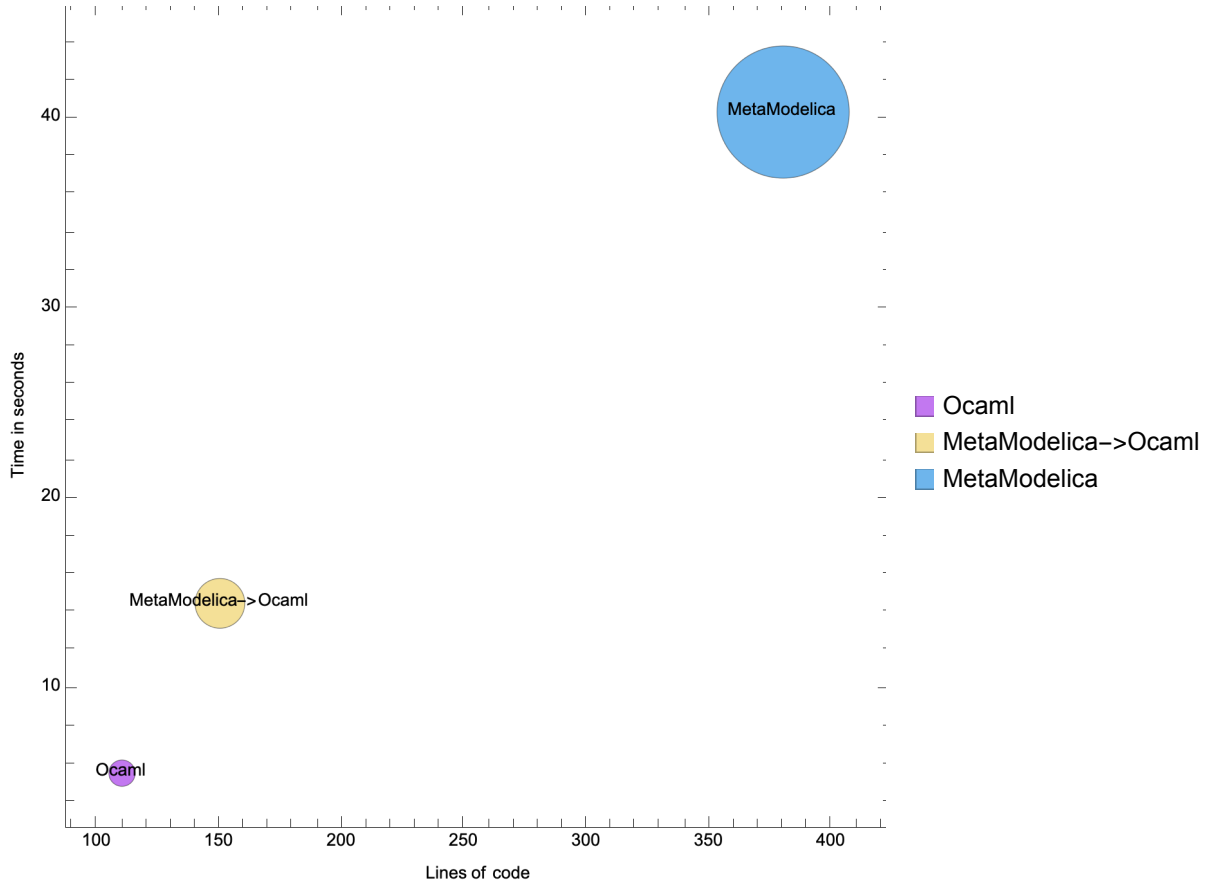
Data conversion

We generated MetaModelica to/from OCaml converters

Performance of converted code

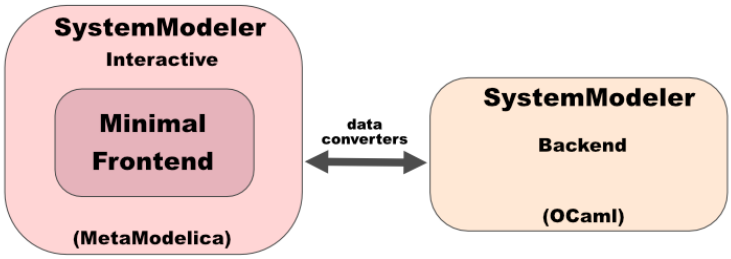
Note: not all optimizations where implemented at this stage.

Out[•]=

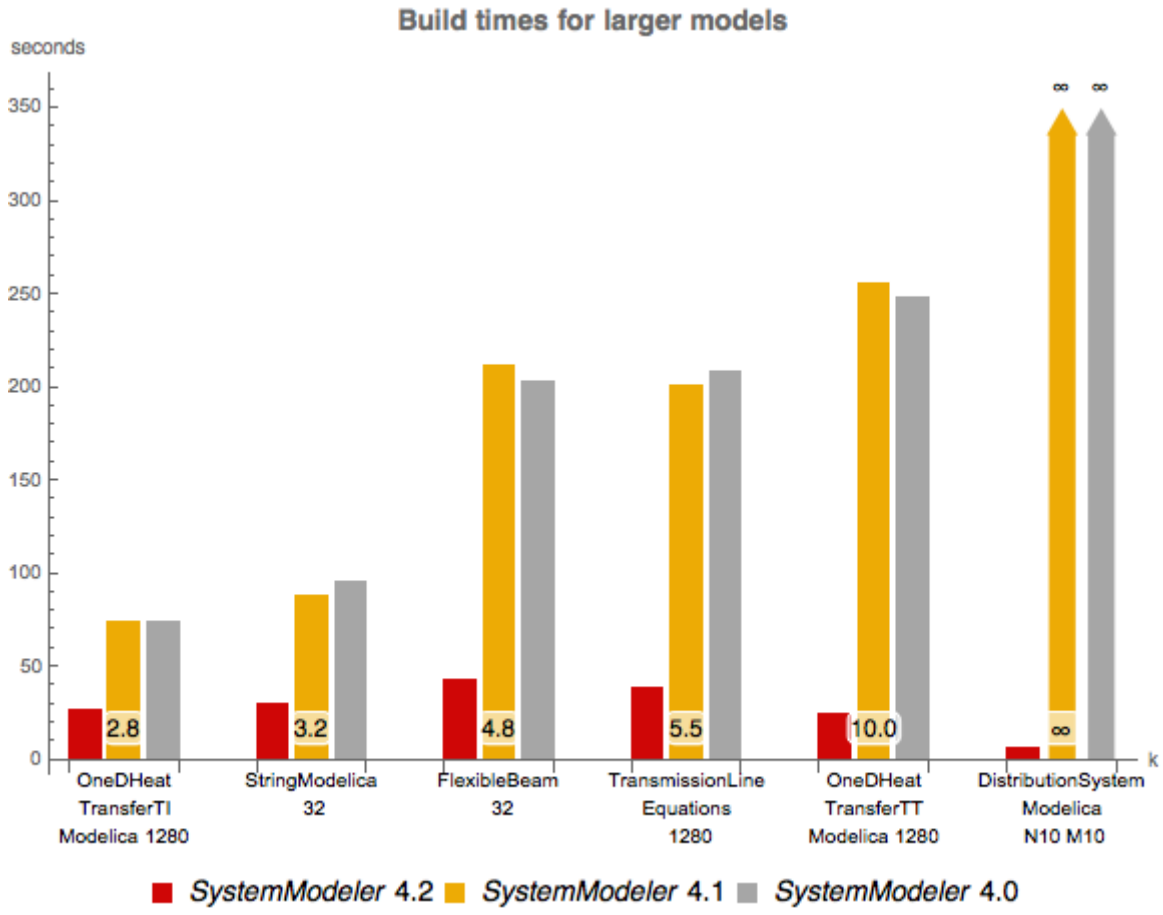


Switching the backend to OCaml

First step



Performance of the mixed Kernel



Other advantages with OCaml

- More warnings
  - Unused variables
  - Unused arguments
  - Uncovered, redundant patter matching cases
- Useful libraries
- Tools
  - Intellisense
  - Time traveling debugger
  - Parser generator
  - Code formatter
- Language features
  - All functional features: anonymous functions, curried functions, if-expressions, nested matching
  - Functors (parametrized modules)
  - Strong typing + type inference
  - Syntax extensions

MetaModelica v2

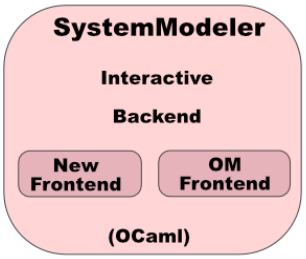
OpenModelica working on a new Frontend using MetaModelica v2

- New code was very imperative (if-statements, for loops)
- Mutability everywhere
- New code was harder to analyze and optimize

Break the dependency

The objectives of SystemModeler and OpenModelica did not align

- Convert all the code (including the frontend library) to OCaml
  - ~510k lines of MetaModelica
  - ~220k lines of OCaml (43%)
- Create our own frontend from scratch
  - Idiomatic OCaml
  - Easier to maintain and fix bugs



Conclusion

- OCaml unleashed the real Functional world
- Performance improvements from 2-10x
- The code is easier to read than before
- Static types help find, fix bugs and focus our testing efforts
- Fixing problems that took more than a week now take days

Thank you!

WolframMathCore.com/careers