Mixed-Signal Verification

Akshay Srivatsan
Áron Ricardo Perez-Lopez
Advisor: Sara Achour

June 7, 2023

Motivation

Mixed-signal systems are ubiquitous and new applications are emerging.

We want to verify them…

…but these systems are hard to verify.

Motivation: Mixed-signal systems

Hardware designs contain both digital and analog components.

Examples: sensors, wireless radios & other communications hardware

Analog components: ADCs, amplifiers, oscillators, filters, etc.

More exotic potential applications: emulate physical systems (e.g., power grid networks) [1]

1. Nagel, Ira, et al. "High-speed power system stability simulation using analog computation: Systematic error analysis." Proceedings of the 17th International Conference Mixed Design of Integrated Circuits and Systems-MIXDES 2010. IEEE, 2010.

Motivation: Verification

Why verify mixed-signal circuits?

1. *Find bugs*: they can cause catastrophic issues; recalls, redesigns and additional tape-outs
2. *Certify correctness*: for compliance purposes, ensuring interoperability, etc.

Status quo: mixed-signal circuits are validated through extensive simulation.

Doesn’t scale well, difficult to find “needle-in-haystack” bugs.

Motivation: Why are mixed-signal circuits hard to verify?

Analog and digital logic are fundamentally different.

Analog: real-valued, continuous-time systems

"Engineering math": Differential equations, transfer functions

Tooling: Analysis (e.g., dReal) and modeling (e.g., Petri nets) tools targeting cyber-physical systems

Digital: combinatorial, clocked systems with digital bits

Discrete math + logic

Tooling: Analysis and modeling tools targeting software and digital hardware

Motivation: Why are mixed-signal circuits hard to verify?

What about directly analyzing both the continuous and discrete parts of mixed-signal circuits from first principles?

Hybrid Systems: Practitioners have devised verification techniques that model mixed-signal systems as hybrid (continuous + discrete) systems then apply hybrid system verification techniques. [1,2]

Typically, these techniques do not scale to large AMS circuits.

1. Radojicic, Carna, et al. "Verification of mixed-signal systems with affine arithmetic assertions." VLSI Design 2013 (2013): 5-5.

2. Dang, Thao, Alexandre Donzé, and Oded Maler. "Verification of analog and mixed-signal circuits using hybrid system techniques." Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings 5. Springer Berlin Heidelberg, 2004.

Basic Idea:

“Reuse” EDA tooling for tool-impoverished domains

Idea: Analog models as digital circuits

Observation*:* There are a lot of scalable analysis tools for digital systems. If we can *emulate* an analog circuit's behavior with a digital circuit, we can use off-the-shelf digital verification tools to verify the mixed-signal system.

Goal: Can we automatically infer a verification-amenable "digital twin", or a digital circuit that emulates an analog circuit's behavior with a digital circuit?

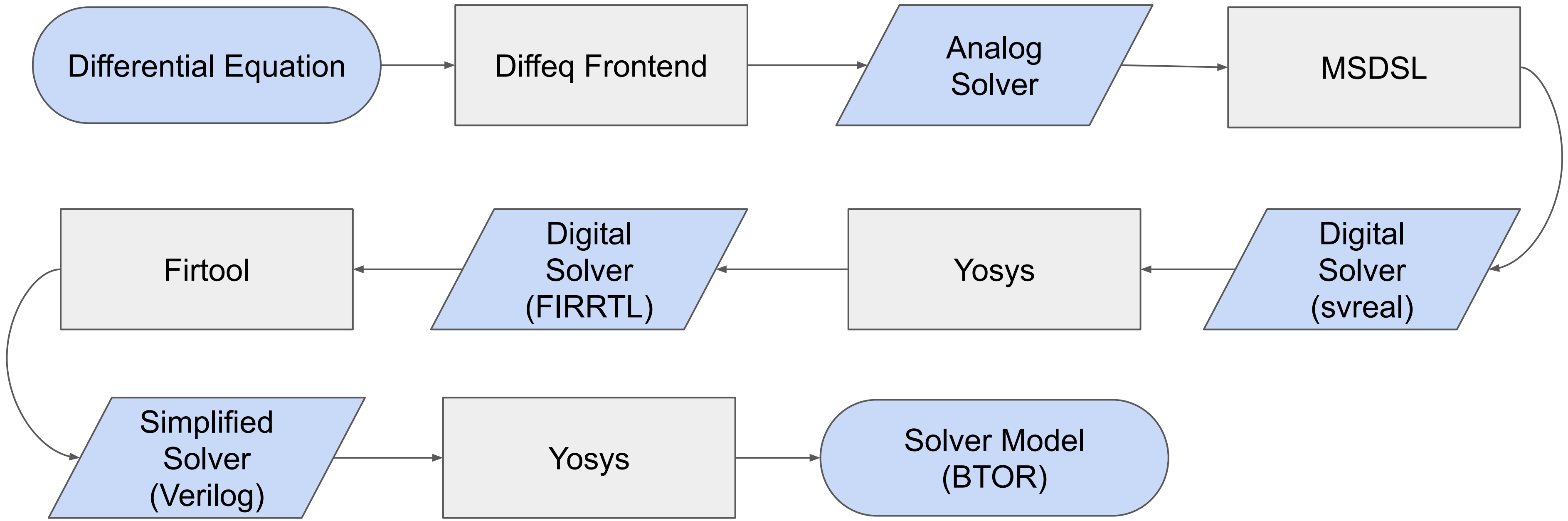
Insight: *We have room to to reduce accuracy.* In a mixed-signal system, analog components are embedded within a digital computing environment:

- Analog inputs and outputs are quantized to digital values.
- Analog circuit state is observed with a clock.

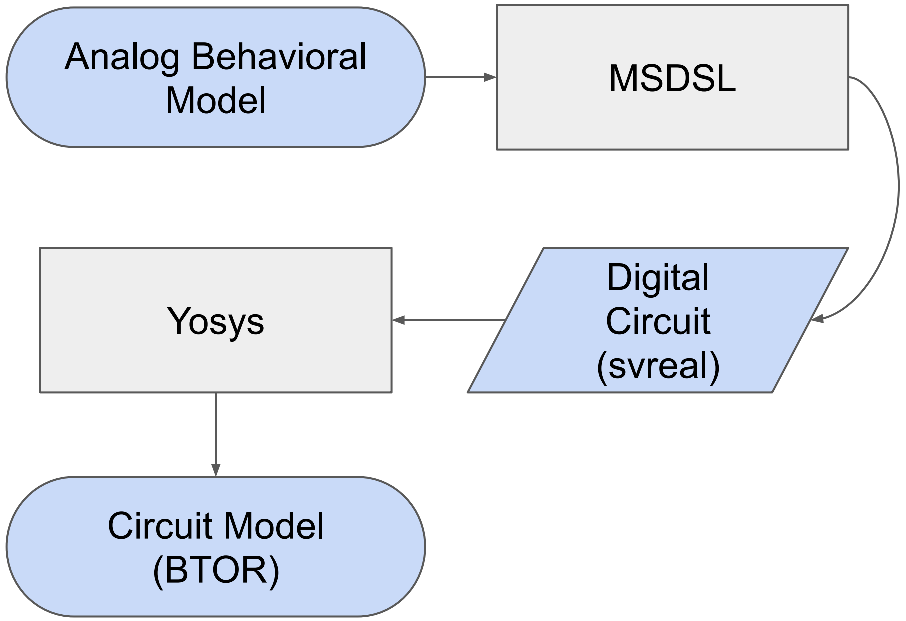
Idea

1. Take an analog design represented by a system of differential equations.
2. Create a digital version that computes the results in discrete time.
3. Transform this circuit into a form that is more amenable to verification.
4. In addition, verify, that the transformation did not change the behavior.
5. ???
6. Profit!

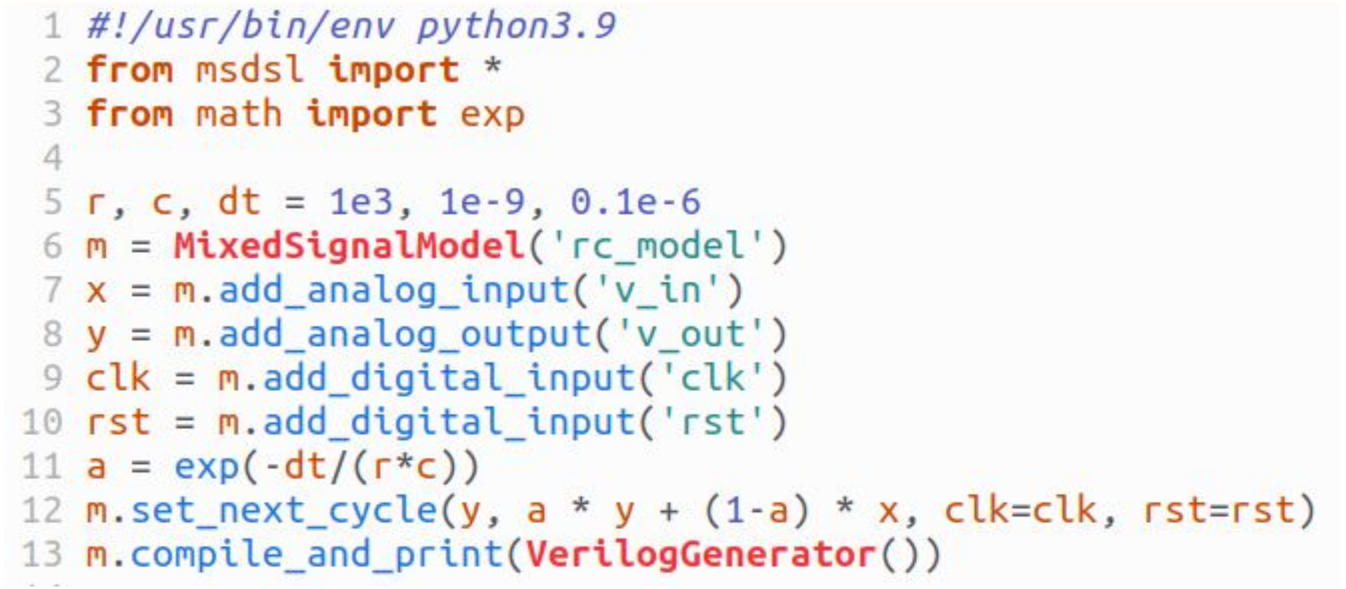
Digital Model Synthesis Flow



Digital Model Synthesis Flow (Proof of Concept)



Analog Behavioral Model (Python + msdsl)

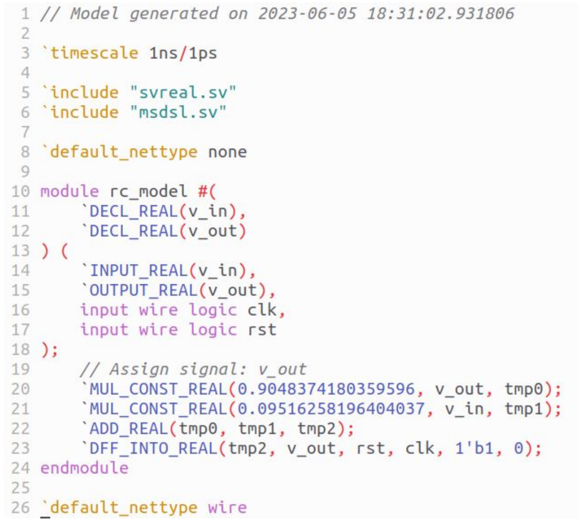


```python
#!/usr/bin/env python3.9
from msdsl import *
from math import exp

r, c, dt = 1e3, 1e-9, 0.1e-6
m = MixedSignalModel('rc_model')
x = m.add_analog_input('v_in')
y = m.add_analog_output('v_out')
clk = m.add_digital_input('clk')
rst = m.add_digital_input('rst')
a = exp(-dt/(r*c))
m.set_next_cycle(y, a * y + (1-a) * x, clk=clk, rst=rst)
m.compile_and_print(VerilogGenerator())
```

Herbst, Steven & Rutsch, Gabriel & Ecker, Wolfgang & Horowitz, Mark. (2021). An Open-Source Framework for FPGA Emulation of Analog/Mixed-Signal Integrated Circuit Designs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. PP. 1-1. 10.1109/TCAD.2021.3102516.

https://github.com/sgherbst/msdsl

Digital implementation of behavioral model (Verilog + svreal)

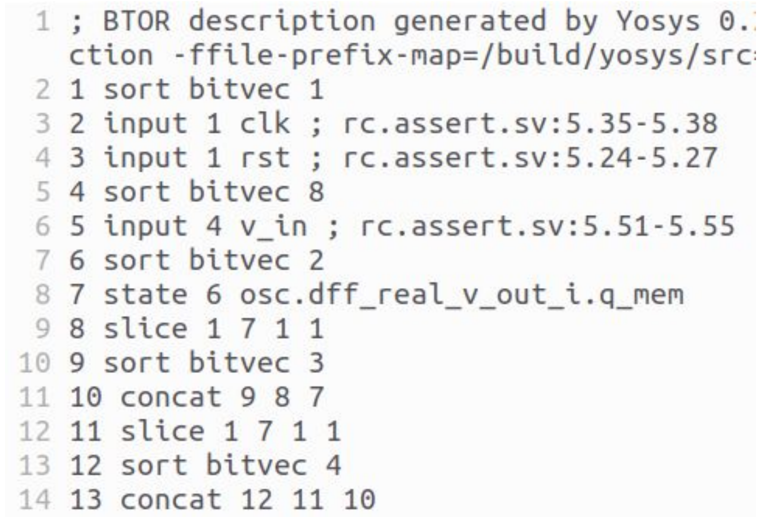


```
// Model generated on 2023-06-05 18:31:02.931806

`timescale 1ns/1ps

`include "svreal.sv"
`include "msdsl.sv"

`default_nettype none

module rc_model #(
    `DECL_REAL(v_in),
    `DECL_REAL(v_out)
) (
    `INPUT_REAL(v_in),
    `OUTPUT_REAL(v_out),
    input wire logic clk,
    input wire logic rst
);
    // Assign signal: v_out
    `MUL_CONST_REAL(0.9048374180359596, v_out, tmp0);
    `MUL_CONST_REAL(0.09516258196404037, v_in, tmp1);
    `ADD_REAL(tmp0, tmp1, tmp2);
    `DFF_INTO_REAL(tmp2, v_out, rst, clk, 1'b1, 0);
endmodule

`default_nettype wire
```

Herbst, Steven & Rutsch, Gabriel & Ecker, Wolfgang & Horowitz, Mark. (2021). An Open-Source Framework for FPGA Emulation of Analog/Mixed-Signal Integrated Circuit Designs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. PP. 1-1. 10.1109/TCAD.2021.3102516.

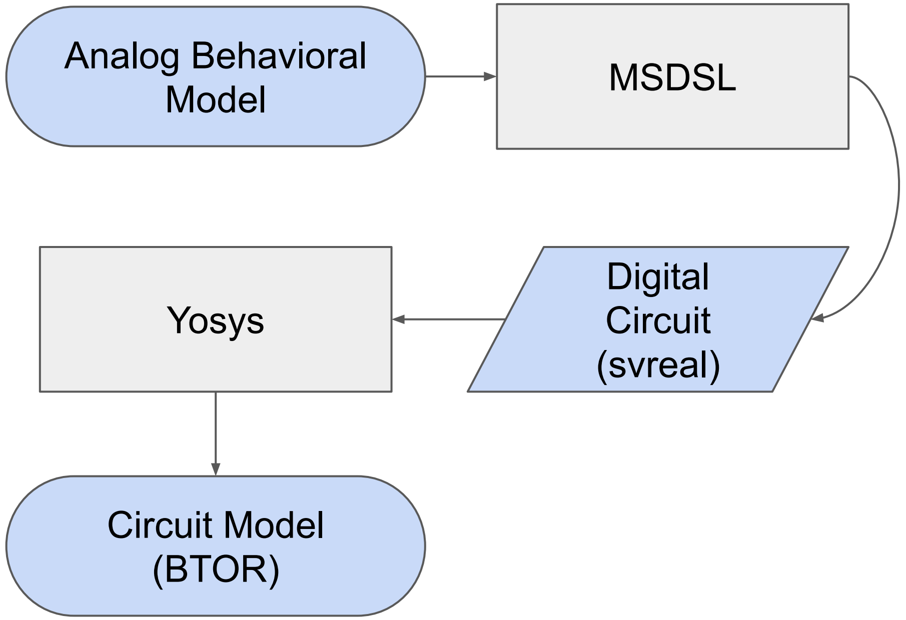
https://github.com/sgherbst/svreal

Digital implementation (model checking format—BTOR)

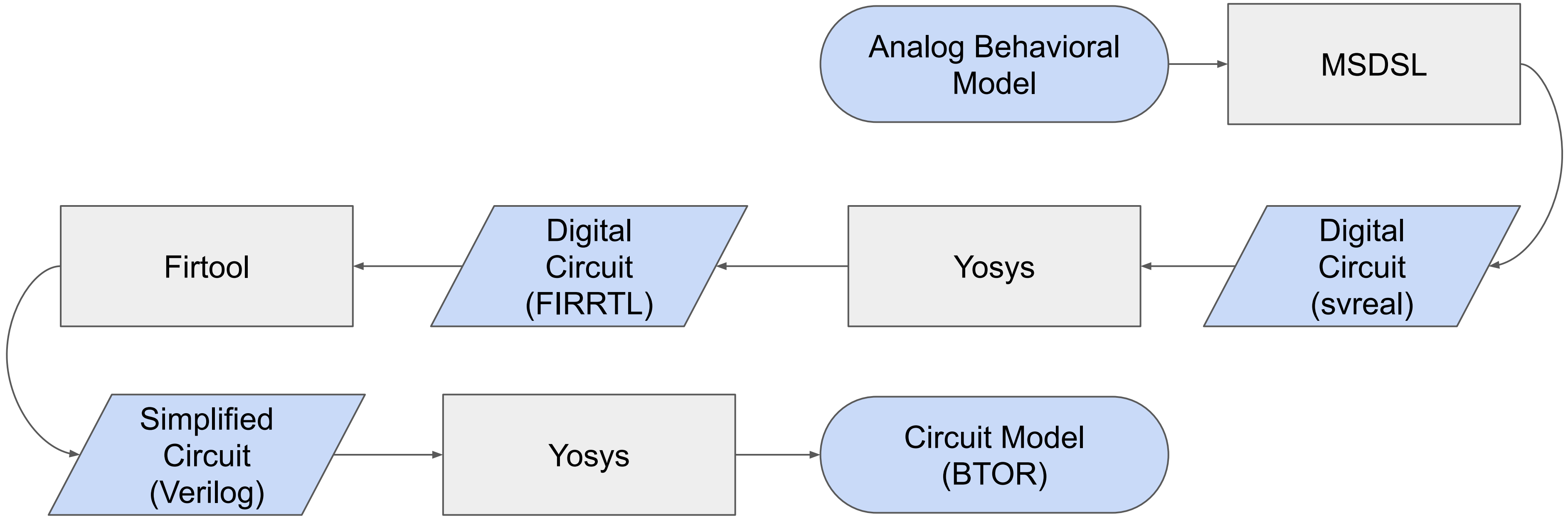


```
; BTOR description generated by Yosys 0.
ction -ffile-prefix-map=/build/yosys/src
1 sort bitvec 1
2 input 1 clk ; rc.assert.sv:5.35-5.38
3 input 1 rst ; rc.assert.sv:5.24-5.27
4 sort bitvec 8
5 input 4 v_in ; rc.assert.sv:5.51-5.55
6 sort bitvec 2
7 state 6 osc.dff_real_v_out_i.q_mem
8 slice 1 7 1 1
9 sort bitvec 3
10 concat 9 8 7
11 slice 1 7 1 1
12 sort bitvec 4
13 concat 12 11 10
```

Digital Model Synthesis Flow (Proof of Concept)



Digital Model Synthesis Flow (+ General Optimization)



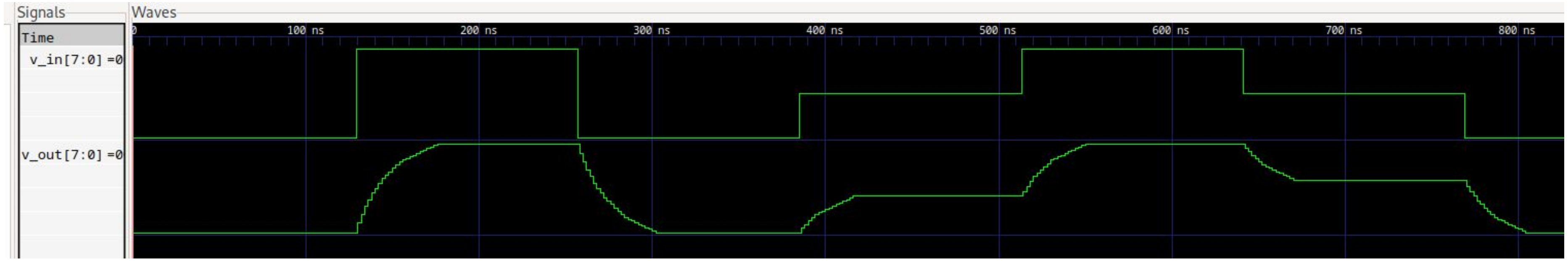
A. Izraelevitz *et al.*, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Irvine, CA, USA, 2017, pp. 209-216, doi: 10.1109/ICCAD.2017.8203780.

General Optimizations via FIRRTL

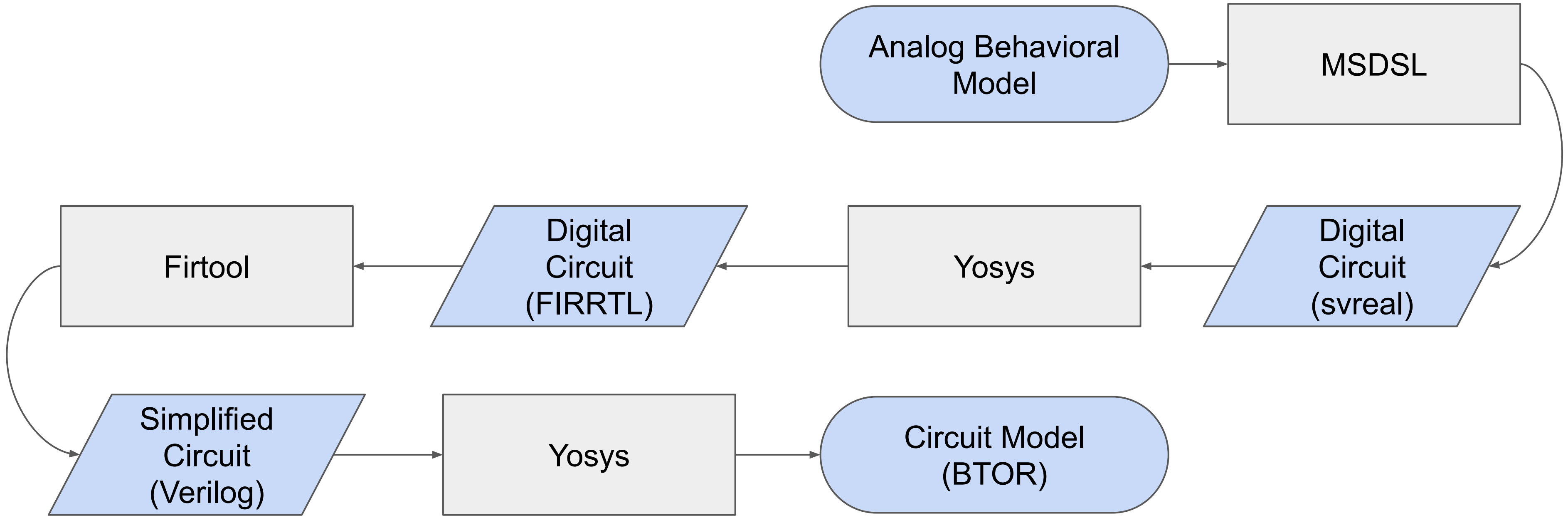
```
Number of wires:                5847
Number of wire bits:            6524
Number of public wires:         42
Number of public wire bits:     719
Number of memories:             0
Number of memory bits:          0
Number of processes:            0
Number of cells:                3771
    $_NAND_                     2539
    $_NOT_                      1224
    $_SDFF_PP0_                 8
```

```
Number of wires:                2893
Number of wire bits:            2984
Number of public wires:         7
Number of public wire bits:     74
Number of memories:             0
Number of memory bits:          0
Number of processes:            0
Number of cells:                1817
    $_NAND_                     1237
    $_NOT_                      572
    $_SDFF_PP0_                 8
```

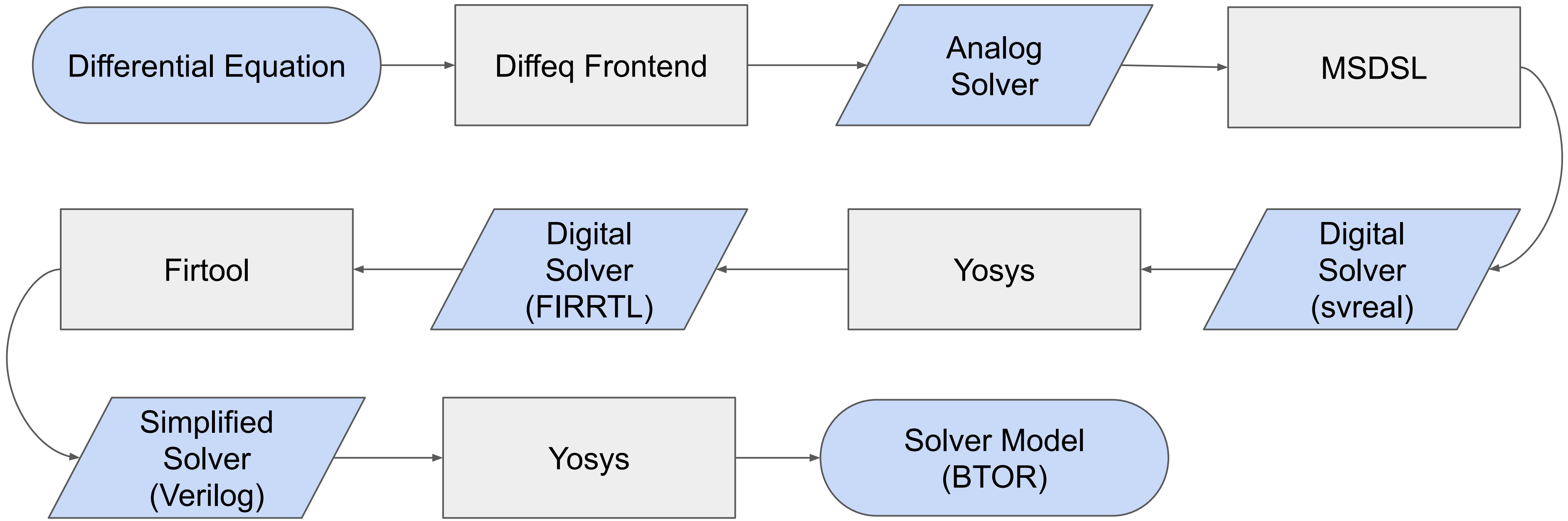
General Optimizations via FIRRTL



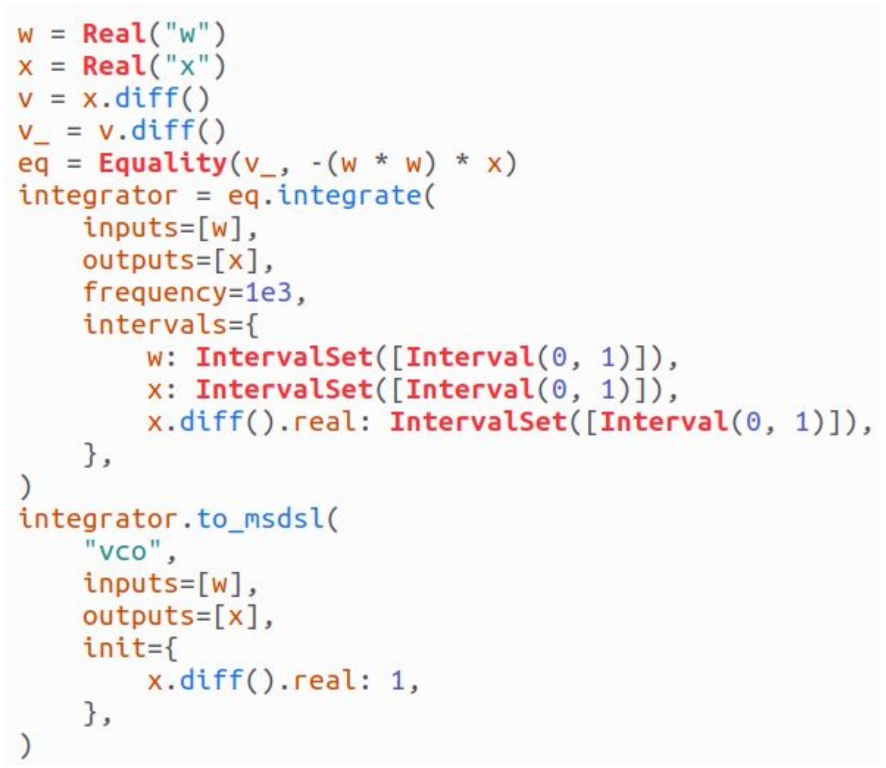
Digital Model Synthesis Flow (+ General Optimization)



Digital Model Synthesis Flow (Complete)



Differential Equation Frontend



```
w = Real("w")
x = Real("x")
v = x.diff()
v_ = v.diff()
eq = Equality(v_, -(w * w) * x)
integrator = eq.integrate(
    inputs=[w],
    outputs=[x],
    frequency=1e3,
    intervals={
        w: IntervalSet([Interval(0, 1)]),
        x: IntervalSet([Interval(0, 1)]),
        x.diff().real: IntervalSet([Interval(0, 1)]),
    },
)
integrator.to_msdsl(
    "vco",
    inputs=[w],
    outputs=[x],
    init={
        x.diff().real: 1,
    },
)
```

Optimizations for Verification

- Goals:
  - minimize the length of the critical path
  - minimize the number of timesteps needed
- Numerical integration is general, but is complicated and needs very narrow timesteps
- In certain cases, we can find a closed-form solution
  - The closed-form solution can be evaluated at an arbitrary timestep, so we don't need to step as frequently
  - The closed-form solution is also simple to evaluate, so it reduces verification complexity

Optimizations for Verification

$$\begin{tikzcd}
\begin{cases} v' = \omega^2 x \\ x' = v \end{cases} \arrow[r] & x'' = \omega^2 x \arrow[dl] \arrow[dr] & \\
\omega' \neq 0 \arrow[d] \arrow[rr, dotted, leftrightarrow, "\text{can switch at ``runtime''}"'] & & \omega' = 0 \arrow[d] \\
\begin{cases} {x''}_{t+1} = \omega^2 x_t \\ {x'}_{t+1} = {x'}_t + {x''}_t \ \mathrm{dt} \\ x_{t+1} = x_t + {x'}_t \ \mathrm{dt} \end{cases} & & x(t) = A\cos(\omega t + B)
\end{tikzcd}$$

*initial values based on closed-form solution

**A* and *B* depend on the previous state of the integrator

Real Number Representation

- svreal supports real numbers in:
  - SystemVerilog real numbers (not synthesizable)
  - Berkeley HardFloat (synthesizable but expensive)
  - Fixed-point numbers (synthesizable)
- Fixed-point numbers can have at most two of:
  - range
  - precision
  - efficiency (low bit-width)
- We annotate values with range and precision information
- The ranges are propagated through the system to find a minimal representation
- We can break up the domain into smaller intervals, each of which can use a different precision

Mixed-signal verification

What do we want to verify?

Equivalence: Once we have an optimized design, we want to make sure that it is equivalent to our reference model.

Notions of equivalence:

- *Strict equality*: not very useful — optimizations might alter behavior
- *Bounded error*: better — allows some leeway
- *Quantization equivalence*: ideal target — more complicated

Mixed-signal verification

How do we verify equivalence?

- **Bounded** model checking:
  - Can show that there are no bugs up to N cycles
  - Can produce a counterexample
  - Useful for bug-finding (to an extent) but for design too!
- **Unbounded** model checking:
  - Can prove that the design is equivalent to the reference model
  - Can (usually) produce a counterexample
  - Much harder problem

Verification flow

1. Embed assertions in a wrapper module that check equivalence of the optimized design and the reference model
2. Yosys emits the top-level design into BTOR(2) format.
   - Represents a transition system
   - State: bit vectors and arrays (à la SMT)
   - Transition relation: next state expressed as a function of current state
3. BTOR2 files can be read by a model checker.
   - We used Pono but, in principle, any other model checker should work.

Preliminary Results

Circuits tested:

- RC filter
- Voltage-controlled oscillator (VCO)

Optimizations tested:

- FIRRTL
- Closed form vs. integrator
- Interval propagation

Assertions tested:

- Equality of outputs
- Difference of outputs (error) less than a threshold

Challenges

Optimization:

- Dividing the domain interval into optimal subdomains
- Dealing with variable timesteps

Verification:

- State space explosion due to unconstrained inputs, wide reals
- Handling designs with multiple clocks

Future Work

- Integrating the optimizer and the verifier
- More optimizations/simplifications
- Using verification results for:
  - Guiding automatic switches between models
  - Finding places were the model can be simplified even further