Cornell University
Computer Systems Laboratory

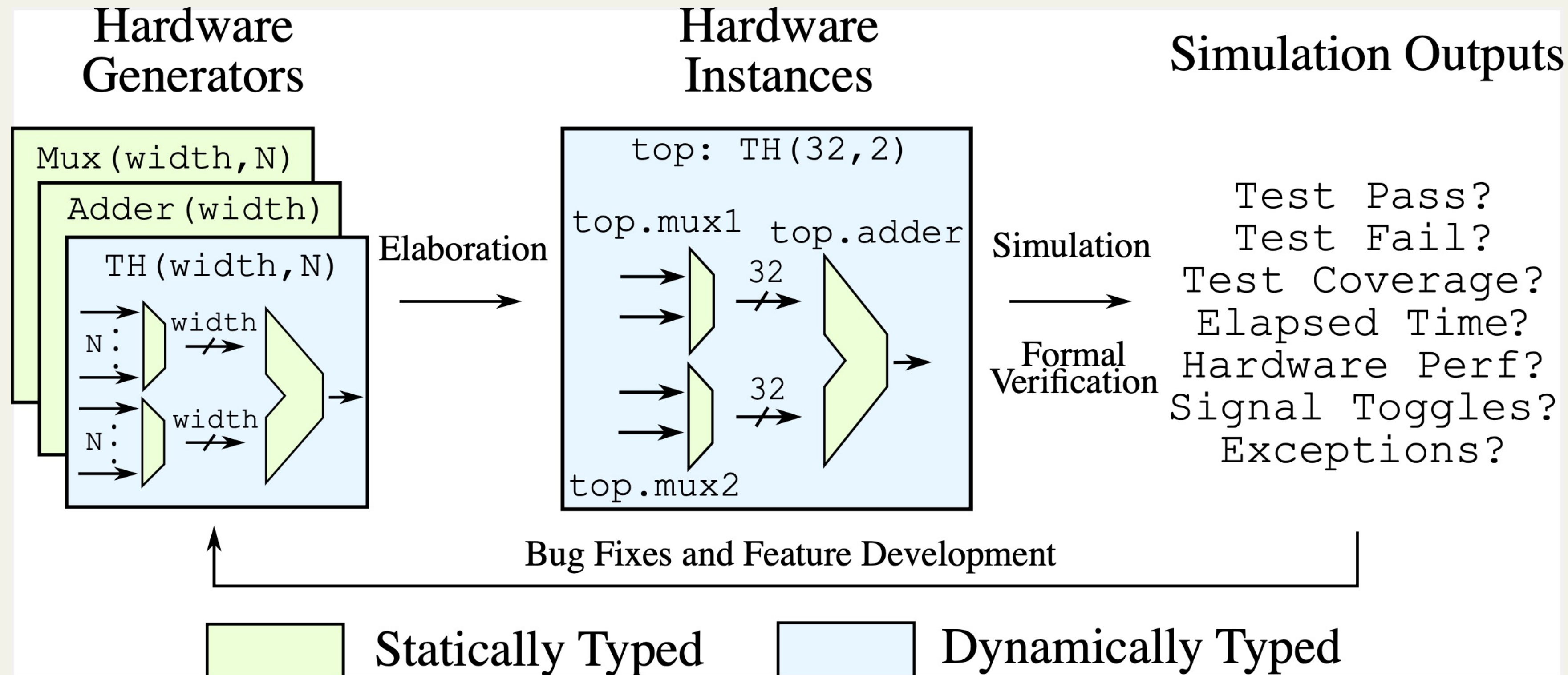
Towards Gradually Typed Hardware Description Languages

Peitian Pan

03/26/2023

A Typical Hardware Development Flow

CSL



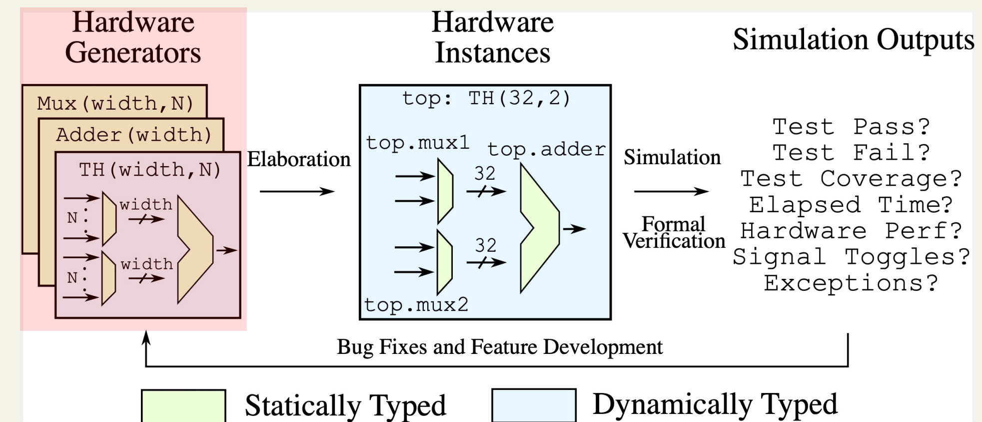
How do existing statically and dynamically typed HDLs accelerate design iterations in this flow?

Statically Typed HDLs: Static Correctness Guarantees

- **Static type checking of generators**
  - Verilog only checks *instances*
  - proves generator invariants across for all possible parameters
  - promotes high-quality generators for better design reuse

```
1  function Bit#(TAdd#(n,1)) adder(
2    Bit#(n) a, Bit#(n) b
3  );
4    Bit#(n) sum; Bit#(TAdd#(n,1)) carry = 0;
5    for (Integer i = 0; i<valueOf(n); i=i+1)
6    begin
7      Bit#(2) tmp = full_adder(a[i], b[i], carry[i]);
8      carry[i+1] = tmp[1]; sum[i] = tmp[0];
9    end
10   return {carry[valueOf(n)],sum};
11 endfunction
```

An Adder Generator in **Bluespec** with
Static Correctness Guarantee on Matching Bitwidths



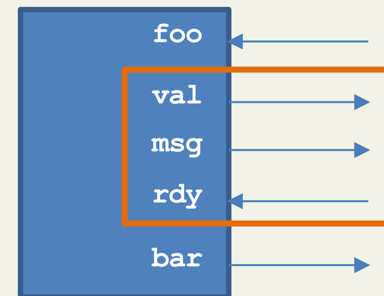
DYNAMICALLY TYPED HDLS: PRODUCTIVE TESTING AND VERIFICATION

CSL

Using PyMTL3 as an example

- Polymorphic test harness
  - » Enables reuse of simulation setup and TB
- Automatic property generation
  - » Enables automatic, blackbox verification

```
1  class ValRdyPropertyGen(BasePass):
2    def apply(s, m):
3      for ifc in m.get_interfaces():
4        if isinstance(ifc, OStreamIfc):
5          s.gen_val_stable_until_rdy(ifc)
```

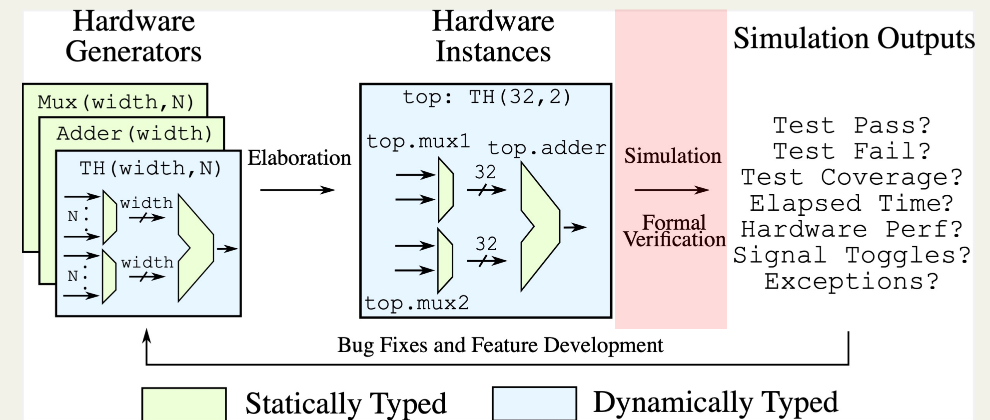



```
assert property(
  val |=> ($stable(val)
s_until_with rdy)
);
```

Automatic Property Generation through Reflection

```
1  class PolyTestHarness:
2    def __init__(s, m, test_vectors, ifunc, ofunc):
3      m.apply(DefaultPassGroup())
4      m.sim_reset()
5      for t in test_vectors:
6        ifunc(m, t)
7        m.sim_eval_combinational()
8        ofunc(m, t)
9        m.sim_tick()
10     print("Test Passed!")
11
12 th = PolyTestHarness(
13   RegAdder(),      [
14     (  3,       5,   "?"),
15     (  1,      42,    8),
16     ( 10,       8,   43),
17     (  0,       0,   18),
18   ],
19   lambda m, t: (assign(m.a, t[0]), assign(m.b, t[1])),
20   lambda m, t: assert_eq(m.out, t[2]) if t[2] != "?" else None)
```

A Polymorphic Test Harness with Customizable Input and Output Functions



The Best of Both Worlds: Gradually Typed HDLs

CSL

Statically Typed HDLs

Gradually Typed HDLs

Dynamically Typed HDLs

+ Static correctness guarantees on generators

+ Fast simulation

- Limited testing/verification productivity

+ Static correctness guarantees

+ High testing/verification productivity

+ Disciplined mixed-typed component composition

+ Simulation performance optimizations

+ High testing/verification productivity

- No static correctness guarantees

- Slow simulation

CSL

- **Using PyMTL3 as an example**
  - Leverage Python type annotation syntax to annotate bitwidths
  - Translate the bitwidth equivalence invariant into integer constraints
  - Use SMT solvers to prove or disprove the invariant

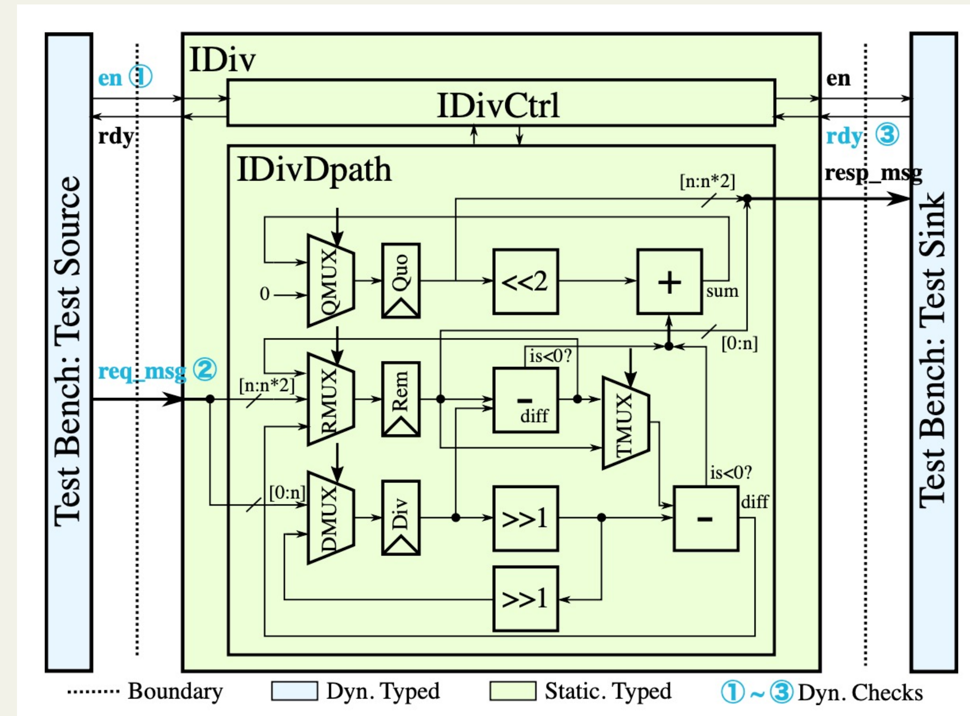
```
1  T_Adder = TypeVar("T_Adder", bound=Bits)
2
3  class Adder(Component, Generic[T_Adder]):
4
5    def __init__(s, Width: Type[T_Adder]) -> None:
6
7    def construct(s, Width: Type[T_Adder]) -> None:
8
9      n = get_nbits(Width)
10
11     # s.a and s.b have type Signal[T_Adder]
12     s.a   = InPort(Width)
13     s.b   = InPort(Width)
14
15     # s.out has type Signal[Bits]
16     s.out = OutPort(mk_bits(n+1))
@update
def upblk() -> None:
  # concat arg1:                  Signal[Bits1]
  # concat arg2:                  Signal[T_Adder]
  # concat(s.carry[n], s.sum):    Signal[Bits]
  s.out @= concat(s.carry[n], s.sum)
```

LHS: n + 1 (from signal definition)
RHS: 1 + n (from semantics of concat and signal definition)

`not ((n+1) == (1+n))` for an integer variable n

GT-HDLs: Safe Mixed-Typed Component Composition

- **The Mixed-Typed Composition Challenge**
  - Statically typed components expect well-typed inputs
  - errors could propagate long past the origin given ill-typed inputs
- **Elaboration-time guards**
  - generators check the given parameters against annotations
- **Simulation-time guards**
  - signal assignments check the given values against its type

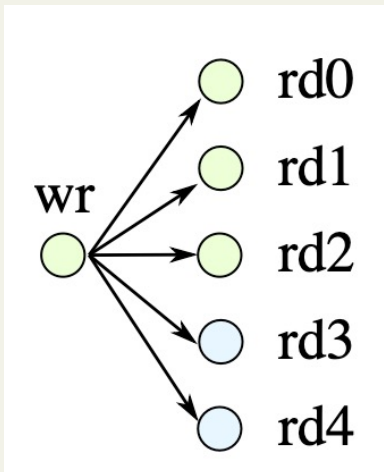


A Mixed-Typed Component Composition with Statically Typed DUT (divider) and Dynamically Typed Test Bench

GT-HDLs: Type-Based Performance Optimizations

CSL

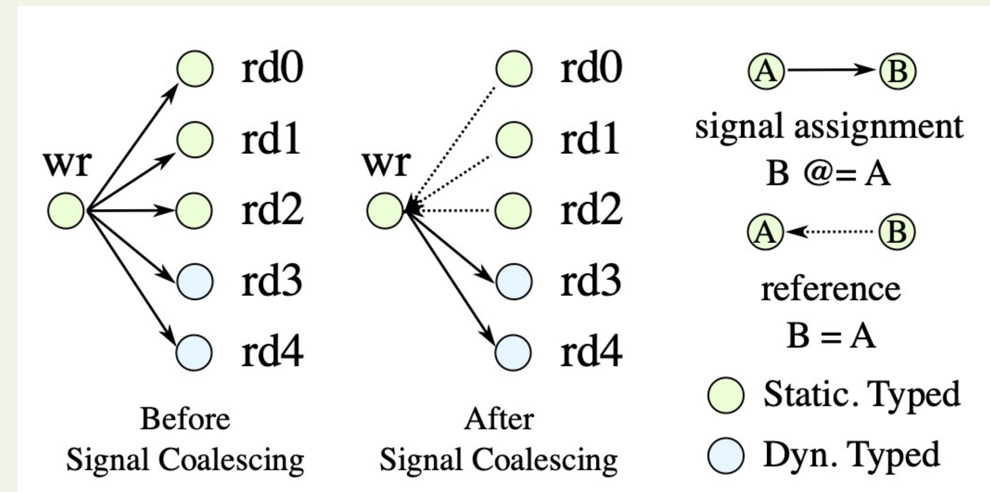
- **Example: signal coalescing**
  - A net data structure is used to represent signal connections
  - Unoptimized: each writer-reader pair needs an assignment every cycle



A net structure of one (1) driver and five (5) readers. Five assignments are needed in every simulated cycle to implement the net behavior.

The unoptimized simulator uses assignment because that's where the simulation-time checks happen.

- Optimized
  - references instead of assignments;
  - assignments still used when simulation guards are required



CONCLUSION

Statically Typed HDLs

Gradually Typed HDLs

Dynamically Typed HDLs

+ **Static correctness guarantees**

+ **High testing/verification productivity**

+ **Disciplined mixed-typed component composition**

+ **Simulation performance optimizations**