Towards Gradually Typed Hardware Description Languages

Peitian Pan
03/26/2023
How do existing statically and dynamically typed HDLs accelerate design iterations in this flow?
### Static Typing of 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

```plaintext
function Bit#(TAdd#(n,1)) adder(
Bit#(n) a, Bit#(n) b
)

Bit#(n) sum; Bit#(TAdd#(n,1)) carry = 0;
for (Integer i = 0; i<valueOf(n); i=i+1)
begin
  Bit#(2) tmp = full_adder(a[i], b[i], carry[i]);
  carry[i+1] = tmp[1]; sum[i] = tmp[0];
end
return {carry[valueOf(n)],sum};
endfunction
```

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

![Diagram showing a generator with static correctness guarantee](image-url)
DYNAMICALLY TYPED HDLS: PRODUCTIVE TESTING AND VERIFICATION

- Using PyMTL3 as an example
  - Polymorphic test harness
    » Enables reuse of simulation setup and TB
  - Automatic property generation
    » Enables automatic, blackbox verification

```
class PolyTestHarness:
    def __init__(s, m, test_vectors, ifunc, ofunc):
        m.apply(DefaultPassGroup())
        m.sim_reset()
        for t in test_vectors:
            ifunc(m, t)
            ofunc(m, t)
        m.sim_tick()
        print("Test Passed!")

th = PolyTestHarness(
    RegAdder(), [
        (3, 4, 5, "?"),
        (1, 42, 8, "?"),
        (10, 8, 43),
        (0, 0, 18),
    ],
    lambda m, t: (assign(m.a, t[0]), assign(m.b, t[1])),
    lambda m, t: assert_eq(m.out, t[2]) if t[2] != "?" else None)
```

A Polymorphic Test Harness with Customizable Input and Output Functions

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

```
assert property(
    val |=> ($stable(val)
          s_until_with_rdy)
)
```

Automatic Property Generation through Reflection

A Polymorphic Test Harness with Customizable Input and Output Functions
### The Best of Both Worlds: Gradually Typed HDLs

<table>
<thead>
<tr>
<th></th>
<th>Static Typing</th>
<th>Gradually Typing</th>
<th>Dynamic Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ Static correctness guarantees on generators</td>
<td>+ Static correctness guarantees</td>
<td>+ High testing/verification productivity</td>
</tr>
<tr>
<td></td>
<td>+ Fast simulation</td>
<td>+ High testing/verification productivity</td>
<td>- No static correctness guarantees</td>
</tr>
<tr>
<td></td>
<td>- Limited testing/verification productivity</td>
<td>+ Disciplined mixed-typed component composition</td>
<td>- Slow simulation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ Simulation performance optimizations</td>
<td></td>
</tr>
</tbody>
</table>
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

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

\[
\text{not } ((n+1) == (1+n)) \text{ for an integer variable } n
\]
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
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.

Before Signal Coalescing

After Signal Coalescing
**CONCLUSION**

- Static correctness guarantees
- High testing/verification productivity
- Disciplined mixed-typed component composition
- Simulation performance optimizations

Gradually Typed HDLs

Statically Typed HDLs

Dynamically Typed HDLs

Gradually Typed HDLs

Statically Typed HDLs

Dynamically Typed HDLs