Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

Peitian Pan, Shunning Jiang, Yanghui Ou, Christopher Batten
MEMOCODE @ Hamburg, Germany
September 22nd, 2023
Motivation: Workflow with Static Hardware Description Languages

Parameterized generators

```
module adder #(parameter W,)
  (input [W-1:0] a,
   input [W-1:0] b,
   output [W:0] c)
);
  assign c = a + b;
endmodule
```

Concrete hardware designs

```
module adder (input [7:0] a,
             input [7:0] b,
             output [8:0] c);
  assign c = a + b;
endmodule
```

Generators → Elaborate → Instances → Create → Simulator

- Static Type Checks
- Instance Checks
- Dynamic Assertions

Ahead of Time Run Time

Executable
Motivation: Workflow with Dynamic Hardware Description Languages

Parameterized generators

```
class Adder(Component):
    def construct(s, W):
        n   = get_nbits(W)
        s.a = InPort (W)
        s.b = InPort (W)
        s.c = OutPort(mk_bits(n+1))

    @update
    def upblk():
        s.c @= s.a + s.b
```

Concrete hardware designs

```
class Adder(Component):
    def construct(s):
        s.a = InPort (Bits8)
        s.b = InPort (Bits8)
        s.c = OutPort(Bits9)

    @update
    def upblk():
        s.c @= s.a + s.b
```
Motivation: Workflow with Dynamic Hardware Description Languages

Parameterized generators

Concrete hardware designs

Optional type checking is quite limited when checking properties ahead-of-time for hardware generators

Research Question: How can we improve the power of ahead-of-time checking in dynamic hardware description languages?

class Adder(Component):
    def construct(s, W):
        n = get_nbits(W)
        s.a = InPort(W)
        s.b = InPort(W)
        s.c = OutPort(mk_bits(n+1))
    @update
    def upblk():
        s.c @= s.a + s.b

class Adder(Component):
    def construct(s):
        s.a = InPort(Bits8)
        s.b = InPort(Bits8)
        s.c = OutPort(Bits9)
    @update
    def upblk():
        s.c @= s.a + s.b

Generators → Elaborate → Instances → Create → Simulator

Optional Type Checks

Instance Checks

Dynamic Assertions
Symbolic elaboration is a **SMT solving-based** static analysis technique that formally verifies many generator properties. Symbolic elaboration:

1. moves checks from *run time* to *ahead of time*,
2. proves properties not just for *one instance*, but for *all instances*, and
3. provides *stronger correctness guarantees*
Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

Motivation

PyMTL3 Framework
Target Generator Properties
Optional Type Checking
Symbolic Elaboration
Evaluation
Python-based hardware modeling, generation, simulation, and verification framework which enables productive multi-level modeling and RTL design.

Hardware Development Workflow with PyMTL3

- **Python**: Functional-Level, Cycle-Level, RTL Simulation, Test Bench
- **SystemVerilog**: generate RTL, co-simulate, synthesize, prototype bring-up

- **FPGA ASIC**
from pymtl3 import *

class RegIncrRTL( Component ):
    def construct( s, N ):
        s.in_ = InPort( N)
        s.out = OutPort( N)
        s.tmp = Wire ( N)
        s.result = Wire ( N)

    @update_ff
    def seq_logic():
        s.tmp <<= s.in_

    @update
    def comb_logic():
        s.result @= s.tmp + 1

    connect( s.result, s.out )

- Hardware modules are Python classes derived from `Component`
- `construct` method for constructing hardware
- `ports` and `wires` for signals
- `update` blocks for modeling combinational and sequential logic
- `connect` for modeling structural connections
PyMTL3 in Academic Chip Tapeouts

TSMC 180nm, 28nm, 16nm; Sky 130nm
GF 130nm, 12nm; Intel 22FFL

Chip Tapeouts Being Tested
HammerBlade OC-FPGA


BRGTC1
IBM 130nm
2x2mm

Celerity
TSMC 16nm
5x5mm

BRGTC2
TSMC 28nm
1x1.25mm

CIFER
GF 12nm
4x4.5mm

BRGTC5
TSMC 180nm
2x2.5mm
Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

**Motivation**

**PyMTL3 Introduction**

**Target Generator Properties**

**Optional Type Checking**

**Symbolic Elaboration**

**Evaluation**

---

**Parameterized Generators**

```python
class Adder(Component):
def construct(s, W):
n = get_nbits(W)
s.a = InPort(W)
s.b = InPort(W)
s.c = OutPort(nk_bits(n+1))
    @update
def upblk():
        s.c @= s.a + s.b
```

**Concrete Hardware Designs**

```python
class Adder(Component):
def construct(s):
s.a = InPort(Bits8)
s.b = InPort(Bits8)
s.c = OutPort(Bits9)
    @update
def upblk():
        s.c @= s.a + s.b
```

---

**Interpreted Bytecode**

**Ahead of Time**

**Run Time**

---

**Generators**

**Elaborate**

**Instances**

**Create**

**Simulator**

**Dynamic Assertions**
Target Property #1: Matching Bitwidths

In a generator, the bitwidths of signals in a structural connection or a binary operation must be the same.

Existing dynamic HDLs only check for matching bitwidths on instances, not generators.

```python
# Instantiate Component
dut1 = NMConn(Bits32, Bits32)
dut1.elaborate() # Pass!

dut2 = NMConn(Bits16, Bits32)
dut2.elaborate() # Fail!

% pytest matching_bitwidths.py -s
E   pymtl3.dsl.errors.InvalidConnectionError: Bitwidth mismatch Bits16 != Bits32
E   - In class <class 'matching_bitwidths.NMConn'>
E   - When connecting s.a <-> s.b
E   Suggestion: make sure both sides of connection have matching bitwidth
```
Target Property #2: Bounded Array Indexing

In a generator, the static indices into arrays must be greater than or equal to 0 and less than the array length.

Existing dynamic HDLs only check for out-of-bound array indices on instances, not generators.

Example bug couldn’t be caught with square reconfigurable arrays.

Reconfigurable Array Example

```python
# Declare Component
class RA(Component):
    def construct(s, r, c, W):
        s.recv_n = [RecvIfc(W) for _ in range(c)]
        s.pes = [PE(W) for _ in range(r * c)]
        for y in range(r):
            for x in range(c):
                if y == r - 1:
                    # Index out-of-range bug; only triggered if r > c
                    # Correct index: y*c+x
                    connect(s.pes[y*r+x].recv[N].msg, s.recv_n[x].msg)

% pytest array_bounds.py -s
> connect(s.proc elems[y*r+x].recv[N].msg, s.recv_n[x].msg)
E    IndexError: list index out of range

# Instantiate Component
dut1 = RA(8, 8, Bits32)
dut1.elaborate() # Pass!
dut2 = RA(16, 8, Bits32)
dut2.elaborate() # Fail!
```
Target Property #2: Bounded Array Indexing

In a generator, the static indices into arrays must be greater than or equal to 0 and less than the array length.

Existing dynamic HDLs only check for out-of-bound array indices on instances, not generators.

Checking the generator not an instance becomes more important with increasing generator complexity.

Example bug couldn't be caught with square reconfigurable arrays.

More details of the other target properties (correct local port direction and valid hierarchical reference) can be found in the paper.

```
% pytest array_bounds.py -s
> connect(s.proc_elems[y*r+x].recv[N].msg, s.recv_n[x].msg)
E    IndexError: list index out of range
```

```
# Instantiate Component
dut1 = RA(8, 8, Bits32)
dut1.elaborate() # Pass!
dut2 = RA(16, 8, Bits32)
dut2.elaborate() # Fail!
```
Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

Motivation

PyMTL3 Framework

Target Generator Properties

Optional Type Checking

Symbolic Elaboration

Evaluation
A Python program may still encounter type errors at run time even if it type checks ahead of time!
Repurposing Mypy for Hardware: PyMTL3 DSL Annotations

```python
class A(Component):
    def construct(s):
        s.in_ = InPort (Bits8)
        s.out = OutPort(Bits8)
        connect(s.in_, s.out)
```

# Hardware Data Types

class Bits: ...
class Bits1(Bits): ...

# Hardware Signal Types

```python
T_SignalN = TypeVar("T_SignalN", bound=Bits)
class Signal(Generic[T_SignalN]):
    def __init__(s, N: Type[T_SignalN]) -> None: ...
    def __and__(s, o: Signal[T_SignalN]) -> Signal[T_SignalN]: ...
    def __or__(s, o: Signal[T_SignalN]) -> Signal[T_SignalN]: ...
    def __xor__(s, o: Signal[T_SignalN]) -> Signal[T_SignalN]: ...
```

InPort = OutPort = Wire = Signal

# Hardware Modeling Primitive Types

```python
T_ConnectN = TypeVar("T_ConnectN", bound=Bits)
def connect(l: Signal[T_ConnectN], r: Signal[T_ConnectN]) -> None: ...
```
class FullAdder(Component):
    def construct(s) -> None:
        s.a = InPort(Bits1)
        s.b = InPort(Bits1)
        s.cin = InPort(Bits1)
        s.sum = OutPort(Bits1)
        s.cout = OutPort(Bits1)
        s.axb = Wire(Bits1)

    @update
    def upblk() -> None:
        s.axb = s.a ^ s.b
        s.sum = s.cin ^ s.axb
        s.cout = (s.axb & s.cin) | s.axb

How does Mypy know the type of ports and wires?
- Mypy infers the type of signals via signature of signal constructors
- Example: s.a has type Signal[Bits1]

How does Mypy verify if bitwidths match for bitwise operators?
- Signal type annotations indicate AND, OR, and XOR take signals of the same bitwidth and return a signal of the same bitwidth

What if there is a bitwidth mismatch for bitwise operators?
- Mypy discovers bitwidth mismatch as a violation of AND, OR, or XOR operator’s type annotations
- Example: s.a = InPort(Bits2) instead of Bits1

% MYPYPATH=/path/to/dsl/annotations mypy full_adder.py
full_adder.py:12: error: Unsupported operand types for ^ ("Signal[Bits2]" and "Signal[Bits1]"
Found 1 error in 1 file (checked 1 source file)
Limitations of Mypy as a Generator Type Checker in Dynamic HDLs

- Fails to reason about \textit{parametrized} bitwidths and \textit{bitwidth-mutating} operations
- Very challenging to encode \textit{array indices} (and also \textit{port directions, hierarchical references})
- In addition, Mypy does not account for \textit{path conditions} (using if-conditions in generators)
Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

Motivation

PyMTL3 Framework

Target Generator Properties

Optional Type Checking

Symbolic Elaboration

Evaluation
Symbolic Elaboration Motivation

How to prove the matching bitwidth property for the assignment in `upblk` in all possible instances?

- Without concrete bitwidths, we **symbolically** determine the bitwidths of assignment LHS/RHS.
  - LHS: bitwidth is \( n+1 \) via `s.out` definition.
  - RHS: bitwidth is the sum of:
    - 1, according to vector indexing semantics.
    - \( n \), according to `s.sum` definition.
- We can translate generator properties into integer constraints to be solved by SMT solvers.
  - Is there a counter example to \((n+1 \neq 1+n)\) for all positive integer \( n \)?
Symbolic Elaboration with an Adder Generator Example

```python
T_AdderW = TypeVar("T_AdderW", bound=Bits)
class Adder(Component, Generic[T_AdderW]):
    def construct(s, W: Type[T_AdderW]) -> None:
        n = get_nbits(W)
        s.a = InPort(W)
        s.b = InPort(W)
        s.out = OutPort(mk_bits(n+1))
        s.carry = Wire(mk_bits(n+1))
        s.sum = Wire(W)
        s.fa = [FullAdder() for _ in range(n)]
        for i in range(n):
            if i >= 0:
                connect(s.carry[i+1], s.fa[i].cout)
        ...

@update
def upblk() -> None:
    s.out @= concat(s.carry[n], s.sum)
```

<table>
<thead>
<tr>
<th>Adder Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adder Abstract Generator Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Symbolic Elaboration with an Adder Generator Example

T_AdderW = TypeVar("T_AdderW", bound=Bits)
class Adder(Component, Generic[T_AdderW]):
    def construct(s, W: Type[T_AdderW]) -> None:
        n = get_nbits(W)
        s.a = InPort(W)
        s.b = InPort(W)
        s.out = OutPort(mk_bits(n+1))
        s.carry = Wire(mk_bits(n+1))
        s.sum = Wire(W)
        s.fa = [FullAdder() for _ in range(n)]
        for i in range(n):
            if i >= 0:
                connect(s.carry[i+1], s.fa[i].cout)
        ...

@update
def upblk() -> None:
    s.out @= concat(s.carry[n], s.sum)
Symbolic Elaboration with an Adder Generator Example

\[ T_{AdderW} = \text{TypeVar("T AdderW", bound=Bits)} \]

```python
class Adder(Component, Generic[T_AdderW]):
    def construct(s, W: Type[T_AdderW]) -> None:
        n       = get_nbits(W)
        s.a     = InPort (W)
        s.b     = InPort (W)
        s.out   = OutPort(mk_bits(n+1))
        s.carry = Wire   (mk_bits(n+1))
        s.sum   = Wire   (W)
        s.fa    = [FullAdder() for _ in range(n)]
        for i in range(n):
            if i >= 0:
                connect( s.carry[i+1], s.fa[i].cout )
        ...

@update
def upblk() -> None:
    s.out @= concat( s.carry[n], s.sum )
```

**Adder Symbol Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type &amp; Metadata</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Bits; generator arg</td>
<td>true</td>
</tr>
<tr>
<td>n</td>
<td>int; get_nbits(W)</td>
<td>true</td>
</tr>
</tbody>
</table>

**Adder Abstract Generator Model**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Definition Condition</th>
</tr>
</thead>
</table>
Symbolic Elaboration with an Adder Generator Example

T_AdderW = TypeVar("T_AdderW", bound=Bits)
class Adder(Component, Generic[T_AdderW]):
    def construct(s, W: Type[T_AdderW]) -> None:
        n = get_nbits(W)
        s.a = InPort(W)
        s.b = InPort(W)
        s.out = OutPort(mk_bits(n+1))
        s.carry = Wire (mk_bits(n+1))
        s.sum = Wire (W)
        s.fa = [FullAdder() for _ in range(n)]
        for i in range(n):
            if i >= 0:
                connect( s.carry[i+1], s.fa[i].cout )
        ...
    @update
    def upblk() -> None:
        s.out @= concat( s.carry[n], s.sum )

Adder Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type &amp; Metadata</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Bits; generator arg</td>
<td>true</td>
</tr>
<tr>
<td>n</td>
<td>int; get_nbits(W)</td>
<td>true</td>
</tr>
</tbody>
</table>

Adder Abstract Generator Model

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>InPort</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>InPort</td>
<td>true</td>
</tr>
</tbody>
</table>
Symbolic Elaboration with an Adder Generator Example

```python
T_AdderW = TypeVar("T_AdderW", bound=Bits)

class Adder(Component, Generic[T_AdderW]):
    def construct(s, W: Type[T_AdderW]) -> None:
        n = get_nbits(W)
        s.a = InPort(W)
        s.b = InPort(W)
        s.out = OutPort(mk_bits(n+1))
        s.carry = Wire(mk_bits(n+1))
        s.sum = Wire(W)
        s.fa = [FullAdder() for _ in range(n)]
        for i in range(n):
            if i >= 0:
                connect(s.carry[i+1], s.fa[i].cout)
        ...

    @update
def upblk() -> None:
        s.out @= concat(s.carry[n], s.sum)
```

Adder Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type &amp; Metadata</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Bits; generator arg</td>
<td>true</td>
</tr>
<tr>
<td>n</td>
<td>int; get_nbits(W)</td>
<td>true</td>
</tr>
</tbody>
</table>

Adder Abstract Generator Model

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>InPort[W]</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>InPort[W]</td>
<td>true</td>
</tr>
<tr>
<td>out</td>
<td>OutPort[n+1]</td>
<td>true</td>
</tr>
<tr>
<td>carry</td>
<td>Wire[n+1]</td>
<td>true</td>
</tr>
</tbody>
</table>
Symbolic Elaboration with an Adder Generator Example

$$T_{\text{Adder}W} = \text{TypeVar}("T_{\text{Adder}W}", \text{bound}=\text{Bits})$$

class Adder(\text{Component}, \text{Generic}[T_{\text{Adder}W}]):
    
def \text{construct}(s, W: \text{Type}[T_{\text{Adder}W}]) \rightarrow \text{None}:
        n = \text{get\_nbits}(W)
        s.a = \text{InPort}(W)
        s.b = \text{InPort}(W)
        s.out = \text{OutPort}(\text{mk\_bits}(n+1))
        s.carry = \text{Wire}(\text{mk\_bits}(n+1))
        s.sum = \text{Wire}(W)
        s.fa = [\text{FullAdder}() \text{ for } _\text{in} \text{ range}(n)]
    
    for i in \text{range}(n):
        if i >= 0:
            \text{connect}(s.carry[i+1], s.fa[i].cout)
    ...

@\text{update}

def upblk() \rightarrow \text{None}:
    s.out @= \text{concat}(s.carry[n], s.sum)
Symbolic Elaboration with an Adder Generator Example

\[ \text{T}\_\text{AdderW} = \text{TypeVar}("T\_\text{AdderW}", \text{bound=Bits}) \]

class Adder(\text{Component}, \text{Generic[T}\_\text{AdderW}]):
    def construct(s, W: \text{Type[T}\_\text{AdderW}]) -> None:
        n = \text{get_nbits}(W)
        s.a = \text{InPort}(W)
        s.b = \text{InPort}(W)
        s.out = \text{OutPort(mk_bits}(n+1))
        s.carry = \text{Wire}(mk_bits}(n+1))
        s.sum = \text{Wire}(W)
        s.fa = \text{[FullAdder() for _ in range}(n)]
        for i in range(n):
            if i >= 0:
                \text{connect}(s.carry[i+1], s.fa[i].cout)
        ...
    @\text{update}
    def upblk() -> None:
        s.out @\text{=} \text{concat}(s.carry[n], s.sum)

\begin{tabular}{|c|c|c|}
\hline
\textbf{Name} & \textbf{Type & Metadata} & \textbf{Definition Condition} \\
\hline
W & Bits; generator arg & true \\
\hline
n & int; get_nbits(W) & true \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
\textbf{Name} & \textbf{Type} & \textbf{Definition Condition} \\
\hline
a & \text{InPort}[W] & true \\
\hline
b & \text{InPort}[W] & true \\
\hline
out & \text{OutPort}[n+1] & true \\
\hline
carry & \text{Wire}[n+1] & true \\
\hline
sum & \text{Wire}[W] & true \\
\hline
fa & \text{List}[\text{FullAdder}] of n & true \\
\hline
\end{tabular}
Symbolic Elaboration with an Adder Generator Example

\[ T_{\text{AdderW}} = \text{TypeVar}("T_{\text{AdderW}}", \text{bound} = \text{Bits}) \]

```python
class Adder(Component, Generic[T_AdderW]):
    def construct(s, W: Type[T_AdderW]) -> None:
        n       = get_nbits(W)
        s.a     = InPort (W)
        s.b     = InPort (W)
        s.out   = OutPort(mk_bits(n+1))
        s.carry = Wire  (mk_bits(n+1))
        s.sum   = Wire  (W)
        s.fa    = [FullAdder() for _ in range(n)]
        for i in range(n):
            if i >= 0:
                connect(s.carry[i+1], s.fa[i].cout)
        ...

@update
def upblk() -> None:
    s.out @ = concat(s.carry[n], s.sum)
```

**Adder Symbol Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type &amp; Metadata</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Bits; generator arg</td>
<td>true</td>
</tr>
<tr>
<td>n</td>
<td>int; get_nbits(W)</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>int; 0 &lt;= i &lt; n</td>
<td>true</td>
</tr>
</tbody>
</table>

**Adder Abstract Generator Model**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>InPort[W]</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>InPort[W]</td>
<td>true</td>
</tr>
<tr>
<td>out</td>
<td>OutPort[n+1]</td>
<td>true</td>
</tr>
<tr>
<td>carry</td>
<td>Wire[n+1]</td>
<td>true</td>
</tr>
<tr>
<td>sum</td>
<td>Wire[n+1]</td>
<td>true</td>
</tr>
<tr>
<td>fa</td>
<td>List[FullAdder] of n</td>
<td>true</td>
</tr>
</tbody>
</table>
Symbolic Elaboration with an Adder Generator Example

Property: matching bitwidths

```
@update
def upblk() -> None:
    s.out @= concat(s.carry[n], s.sum)
```

LHS: n+1, according to abstract model (s.out)
RHS: 1+W, according to concat semantics and abstract model (s.sum)

Use condition: true (not under if-else statements)

Integer constraint:
- \( \neg (n + 1 = 1 + W) \land true \land true \land (n = W) \)
  - true \land true: from use condition and definition condition

SMT solver finds the constraint unsatisfiable => proof!

<table>
<thead>
<tr>
<th>Adder Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>W</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adder Abstract Generator Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>out</td>
</tr>
<tr>
<td>carry</td>
</tr>
<tr>
<td>sum</td>
</tr>
<tr>
<td>fa</td>
</tr>
</tbody>
</table>
Symbolic Elaboration with an Adder Generator Example

Property: bounded array indexing

```python
for i in range(n):
    if i >= 0:
        connect(s.carry[i+1], s.fa[i].cout)
```

Array length: n+1, according to abstract model (s.carry)
Index expression: i+1

Use condition: i >= 0

Integer constraint:
- ¬(0 <= i+1 < n+1) ∧ (i >= 0) ∧ true ∧ (0 <= i < n)
  - (i >= 0) ∧ true: from use condition and definition condition
  - (0 <= i < n): from loop induction variable i

SMT solver finds the constraint unsatisfiable => proof!

---

### Adder Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type &amp; Metadata</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Bits; generator arg</td>
<td>true</td>
</tr>
<tr>
<td>n</td>
<td>int; get_nbits(W)</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>int; 0 &lt;= i &lt; n</td>
<td>true</td>
</tr>
</tbody>
</table>

---

### Adder Abstract Generator Model

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>InPort[W]</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>InPort[W]</td>
<td>true</td>
</tr>
<tr>
<td>out</td>
<td>OutPort[n+1]</td>
<td>true</td>
</tr>
<tr>
<td>carry</td>
<td>Wire[n+1]</td>
<td>true</td>
</tr>
<tr>
<td>sum</td>
<td>Wire[W]</td>
<td>true</td>
</tr>
<tr>
<td>fa</td>
<td>List[FullAdder] of n</td>
<td>true</td>
</tr>
</tbody>
</table>
Symbolic Elaboration with an Adder Generator Example

Property: bounded array indexing

```python
for i in range(n):
    if i >= 0:
        connect( s.carry[i+1], s.fa[i].cout )
```

Array length: n+1, according to abstract model (s.carry)

Index expression: i+1

Use condition: i >= 0

Integer constraint:
• \( \neg (0 \leq i+1 < n+1) \land (i \geq 0) \land \text{true} \land (0 \leq i < n) \)
  • (i >= 0) \land \text{true}: \text{from use condition and definition condition}
  • (0 <= i < n): \text{from loop induction variable i}

SMT solver finds the constraint unsatisfiable => proof!

---

## Adder Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type &amp; Metadata</th>
<th>Definition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Bits; generator arg</td>
<td>true</td>
</tr>
<tr>
<td>n</td>
<td>int; get_nbits(W)</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>int; 0 &lt;= i &lt; n</td>
<td>true</td>
</tr>
</tbody>
</table>

### Adder Abstract Generator Model

- **a**: InPort[W]
- **b**: InPort[W]
- **out**: OutPort[n+1]
- **carry**: Wire[n+1]
- **sum**: Wire[W]
- **fa**: List[FullAdder] of n

More details about the symbolic elaboration algorithm, the supported syntax, and the implementation can be found in the paper.
Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

Motivation

PyMTL3 Framework

Target Generator Properties

Optional Type Checking

Symbolic Elaboration

Evaluation
**Evaluation Methodology: Evaluated Schemes**

- **PyMTL3 (P)**
  - All checks are on a single instance and happen at run time
  - Uses instance checks to identify structural design bugs
  - Uses dynamic assertions from a testbench to identify functional design bugs

- **PyMTL3 + Mypy (M)**
  - Optional type checking on generators ahead of time
  - Still perform all instance checks and dynamic assertions at run time

- **PyMTL3 + Symbolic Elaboration (S)**
  - Symbolic elaboration on generators ahead of time
  - Still perform all instance checks and dynamic assertions at run time
### Evaluation Methodology: Generators used in Experiments

<table>
<thead>
<tr>
<th>Generator</th>
<th>Generator Lines of Code</th>
<th>Instance Lines of Code</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFSR</td>
<td>31</td>
<td>23</td>
<td>32-bit register</td>
</tr>
<tr>
<td>Gray Encoder/Decoder</td>
<td>42</td>
<td>76</td>
<td>32-bit input</td>
</tr>
<tr>
<td>Priority Encoder</td>
<td>45</td>
<td>79</td>
<td>32-bit input</td>
</tr>
<tr>
<td>Round-Robin Arbiter</td>
<td>49</td>
<td>88</td>
<td>4 requesters</td>
</tr>
<tr>
<td>FIFO</td>
<td>125</td>
<td>174</td>
<td>32-bit 2-element</td>
</tr>
<tr>
<td>Divider</td>
<td>172</td>
<td>535</td>
<td>32-bit data path</td>
</tr>
<tr>
<td>Processor</td>
<td>903</td>
<td>2135</td>
<td>Single-core RV32-IM</td>
</tr>
<tr>
<td>CGRA</td>
<td>1170</td>
<td>4004</td>
<td>8x8 32-bit PE array</td>
</tr>
</tbody>
</table>
**Evaluation Methodology: AST Fuzzer**

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Example</th>
<th>Possible Property Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwidth Mutation</td>
<td>Bits32 -&gt; Bits31</td>
<td>Matching bitwidth</td>
</tr>
<tr>
<td>Component Attribute Mutation</td>
<td>s.msg -&gt; s.val</td>
<td>Valid hierarchical reference</td>
</tr>
<tr>
<td>Port Direction Mutation</td>
<td>InPort -&gt; OutPort</td>
<td>Correct local port direction</td>
</tr>
<tr>
<td>Name Expression Mutation</td>
<td>s.fa = [FullAdder() for _ in range(n)] -&gt; s.fa = [Xor() for _ in range(n)]</td>
<td>Valid hierarchical reference, correct local port direction</td>
</tr>
<tr>
<td>Attribute Base Mutation</td>
<td>s.out -&gt; out</td>
<td>N/A; tests the robustness of SE implementation</td>
</tr>
<tr>
<td>Functionality Mutation</td>
<td>s.carry[n+1] -&gt; s.carry[n-1]</td>
<td>Bounded array indexing</td>
</tr>
</tbody>
</table>

AST fuzzer may inject benign mutations – a mutated generator is not always bugged!
Evaluation: Bug Detection Results

- P: PyMTL3
- M: PyMTL3 with MyPy
- S: PyMTL3 with Symbolic Elaboration

The chart shows the percentage of bugs detected for different components (LFSR, Gray, Priority Encoder, Arbiter, FIFO, Divider, Processor, CGRA) under the three configurations. The bars are color-coded for Ahead of Time (red) and Run Time (blue) detections.
Evaluation: Bug Detection Results

(1) Moves checks to ahead of time
(2) Proves property for all instances instead of one instance
(3) Provides stronger correctness guarantees
Related Work

- **Symbolic execution** [Schwartz’10, Cadar’11, Stephens’16]
  - Static analysis technique to discover vulnerabilities in software programs
  - Our work creatively adopts symbolic execution for elaboration in dynamic HDLs

- **PL type system research** [Nikhil’04, Rondon’08]
  - Bluespec adopts *numeric types* to (partially) address bitwidth mismatches
  - *Liquid types* leverages constraint solving to detect out-of-bound indices
  - Our work builds off this prior work, targets dynamic HDLs with more properties

- **Verilog generator consistency checks** [Gillenwater’08, Salama’11]
  - Featherweight Verilog: an SMT solving-based static analysis technique for structural connection errors
  - Our work targets generators in dynamic HDLs and supports more properties
Symbolic elaboration is an SMT solving-based static analysis technique that:

- moves run time checks to ahead of time;
- checks not just one instance, but all instances; and
- provides strong correctness guarantees.