Cornell University
Computer Systems Laboratory

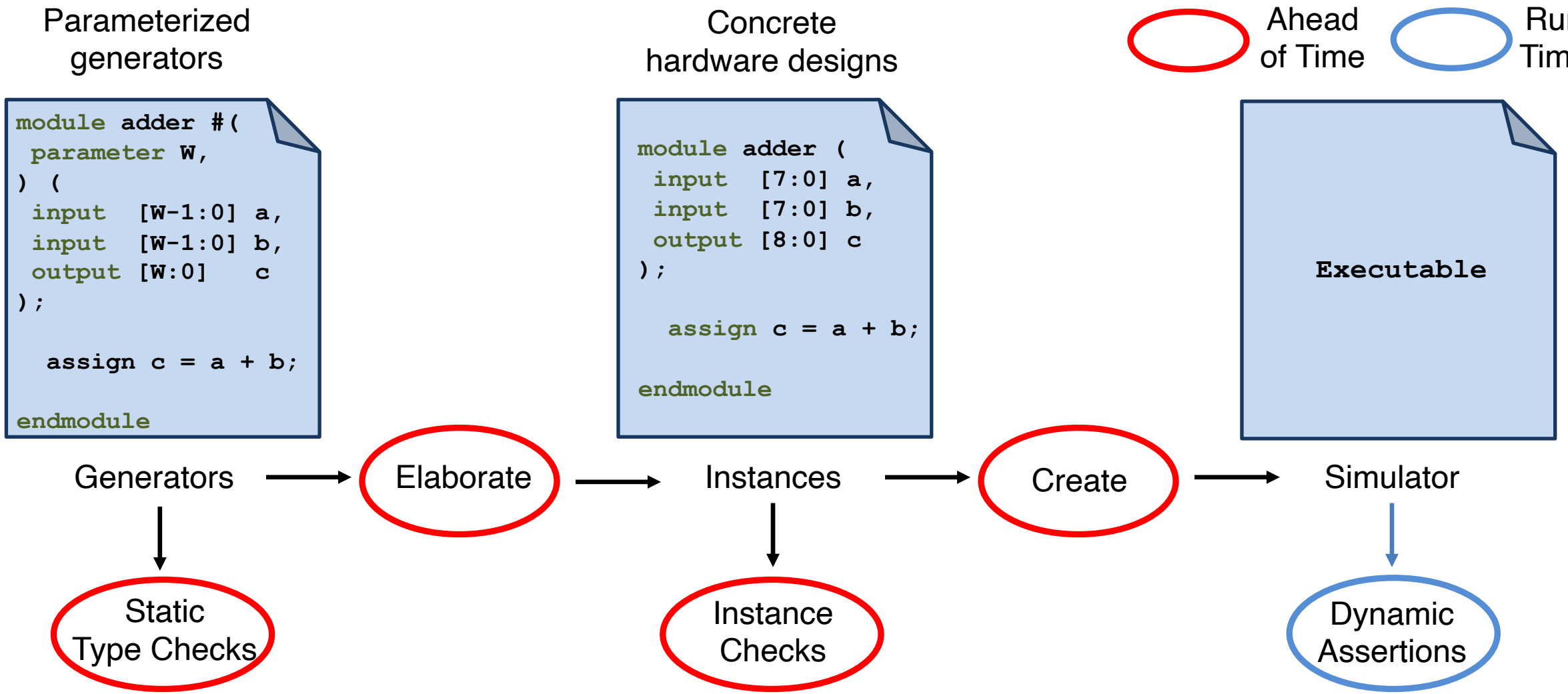
Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

Peitian Pan, Shunning Jiang, Yanghui Ou, **Christopher Batten**

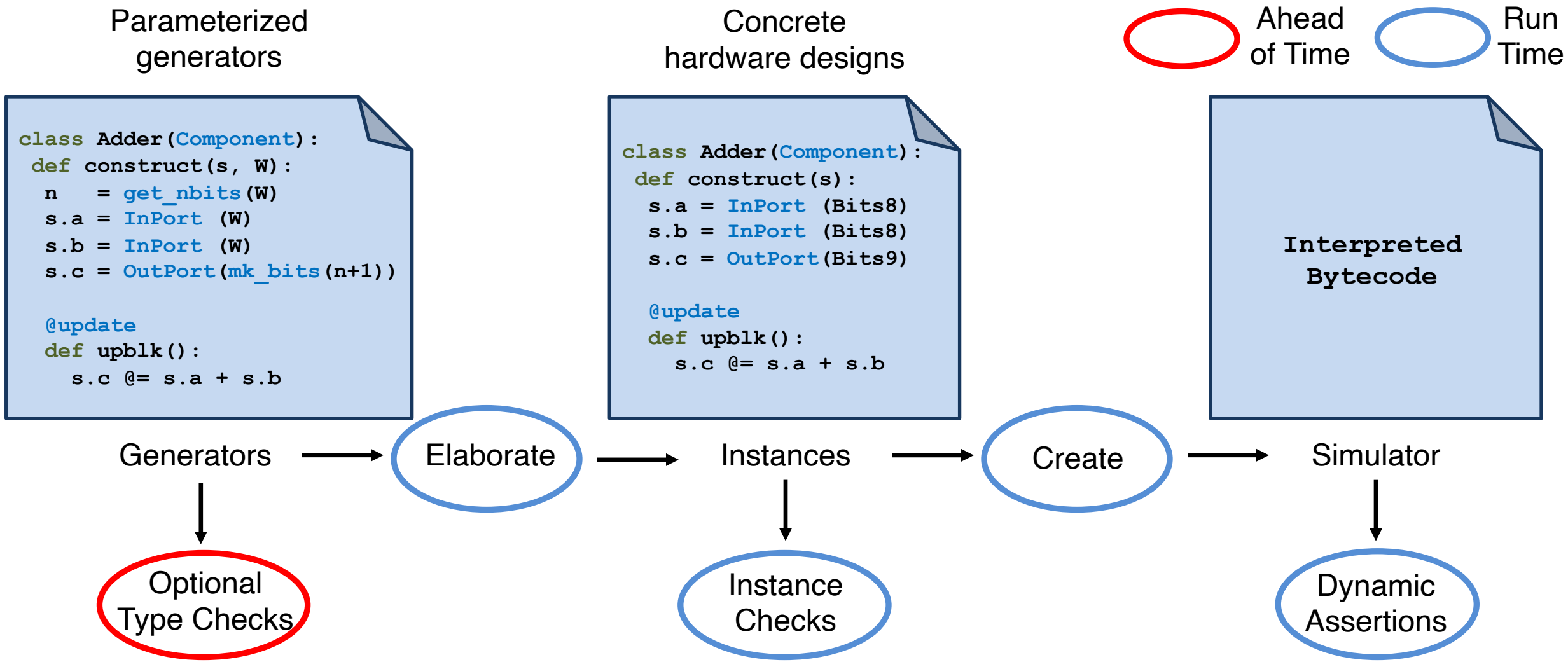
MEMOCODE @ Hamburg, Germany

September 22nd, 2023

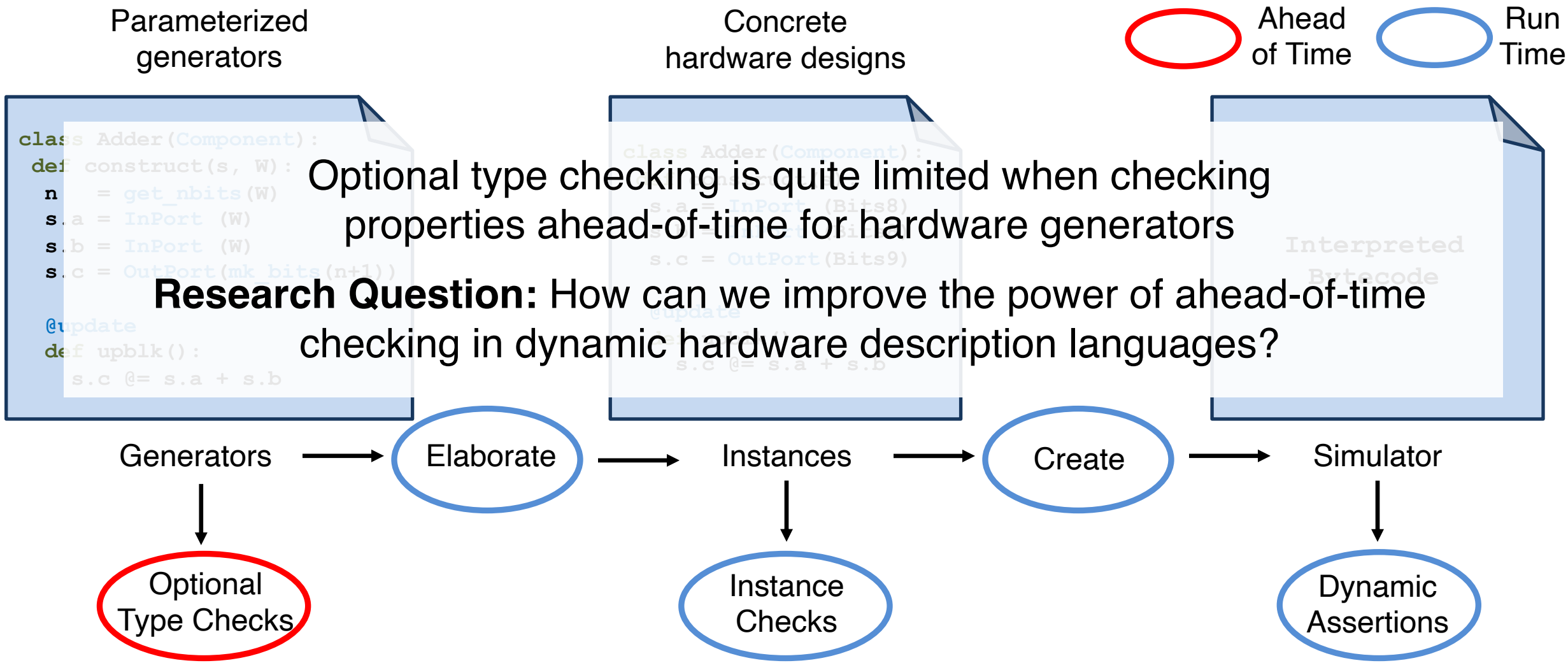
Motivation: Workflow with Static Hardware Description Languages



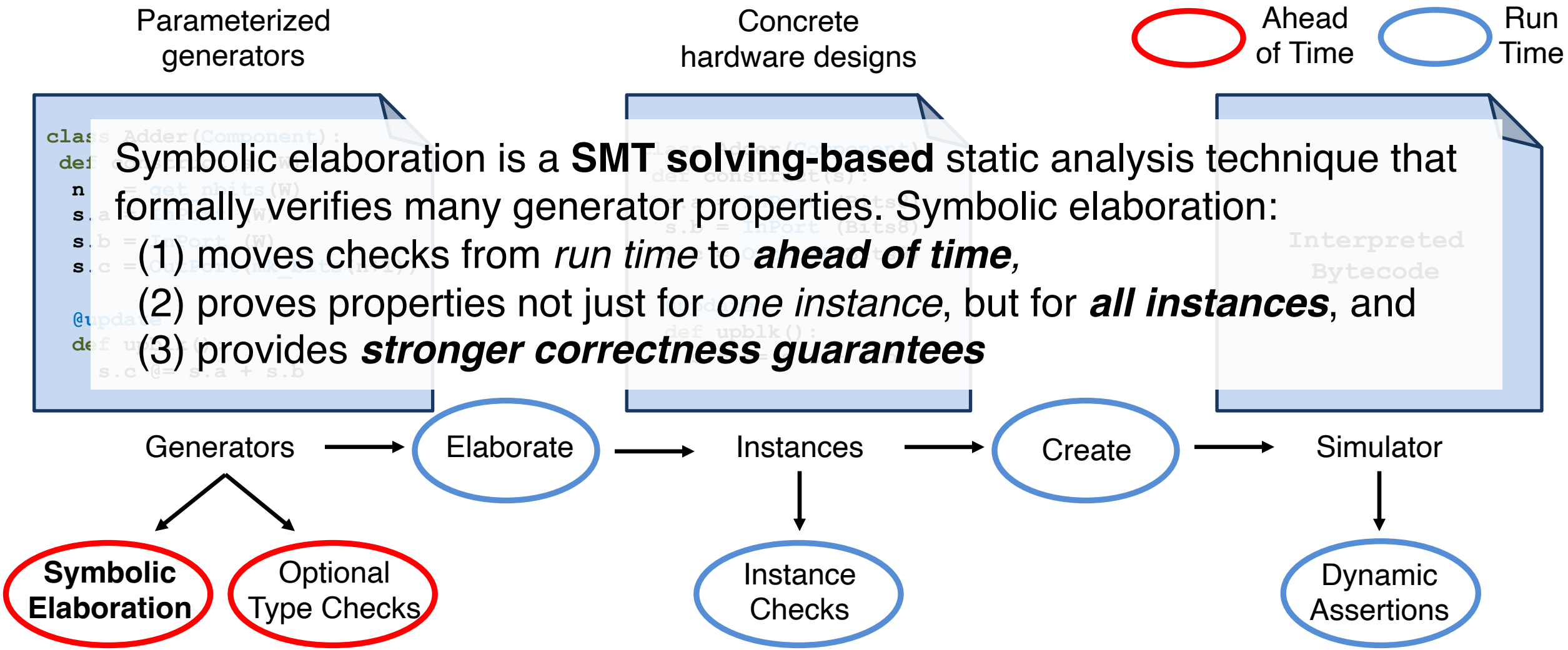
Motivation: Workflow with Dynamic Hardware Description Languages



Motivation: Workflow with Dynamic Hardware Description Languages



Motivation: Workflow with Dynamic Hardware Description Languages



Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

Motivation

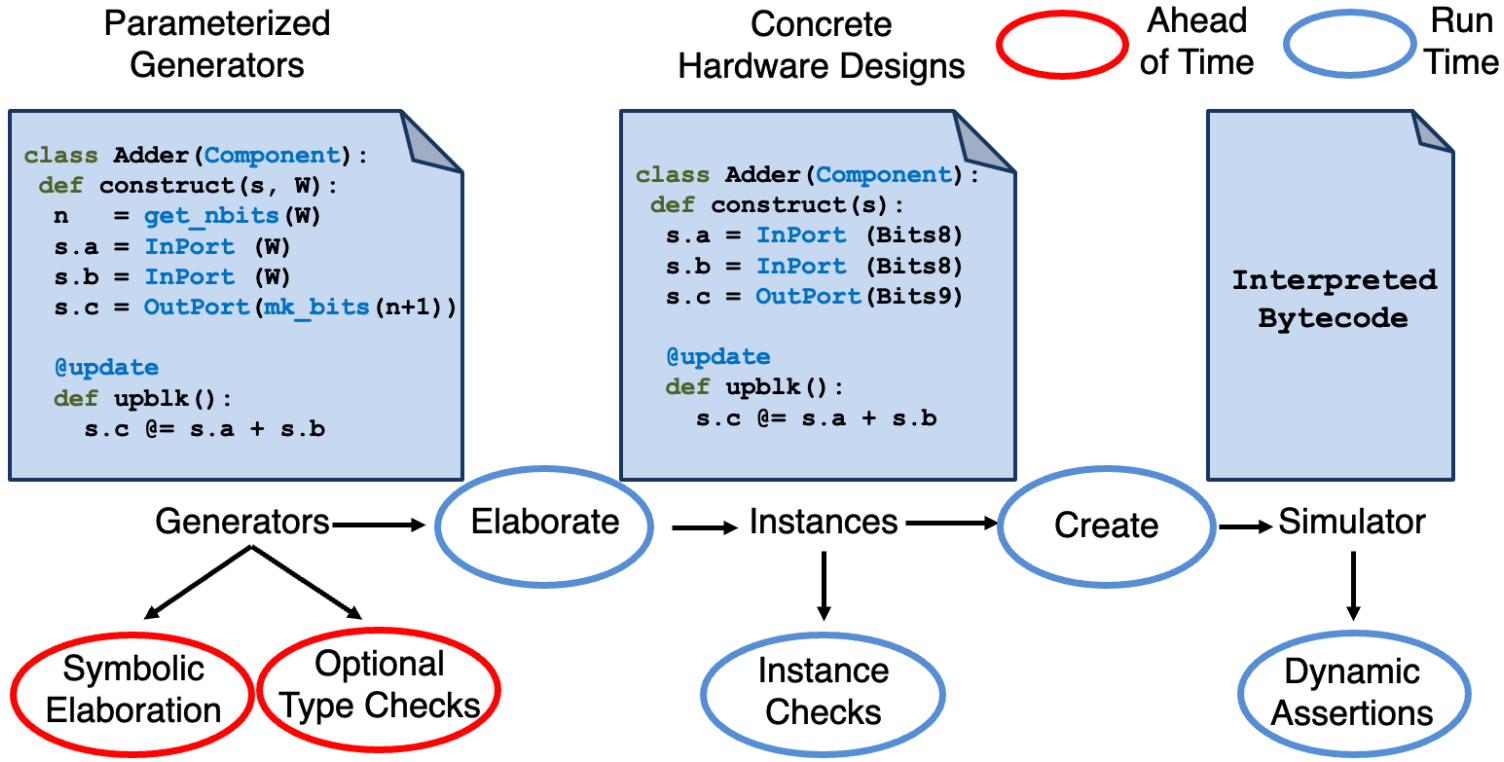
PyMTL3 Framework

Target Generator Properties

Optional Type Checking

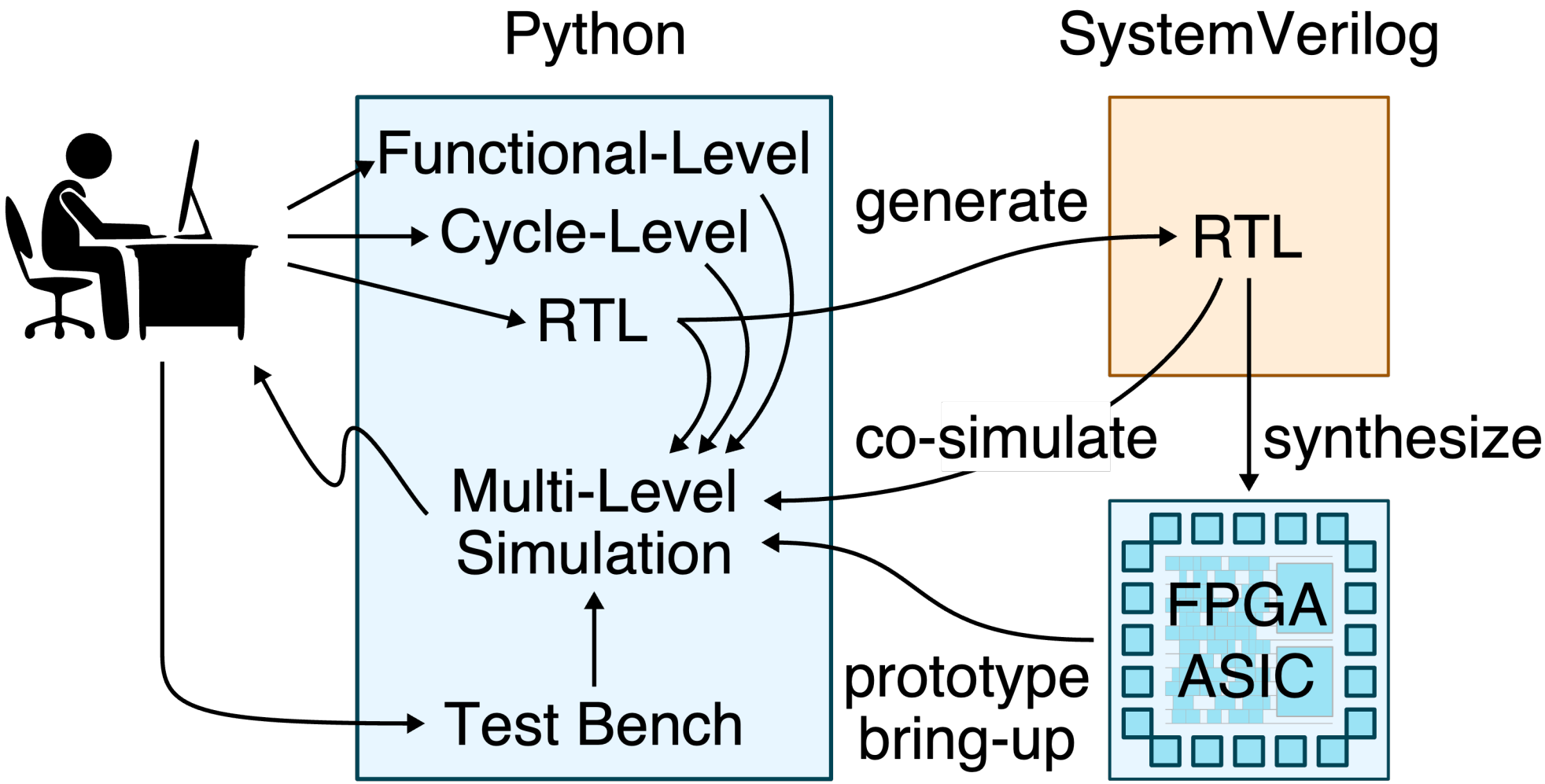
Symbolic Elaboration

Evaluation



Hardware Development Workflow with PyMTL3

Python-based hardware **modeling**, **generation**, **simulation**, and **verification** framework which enables productive **multi-level modeling** and **RTL design**



RTL Modeling Example in PyMTL3

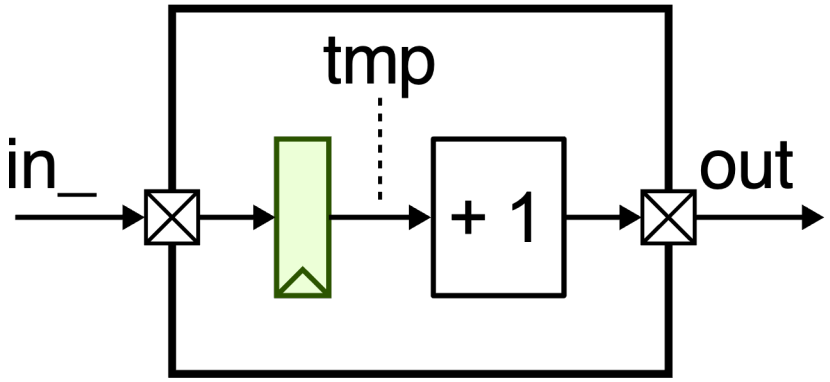
```
from pymtl3 import *

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

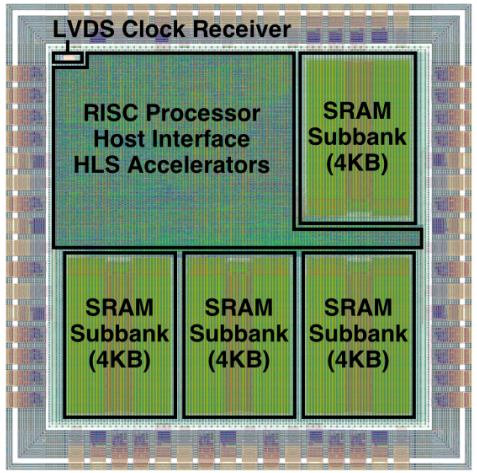
PyMTL3 in Academic Chip Tapeouts

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

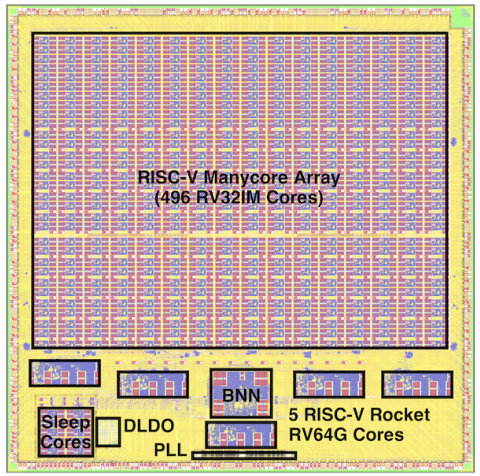
BRGTC3&4
TSMC 180nm
2x2.5mm

Chip Tapeouts Being Tested
HammerBlade **OC-FPGA**

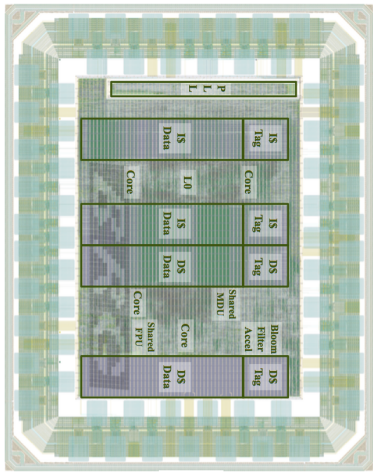
2015 2016 2017 2018 2019 2020 2021 2022 2023



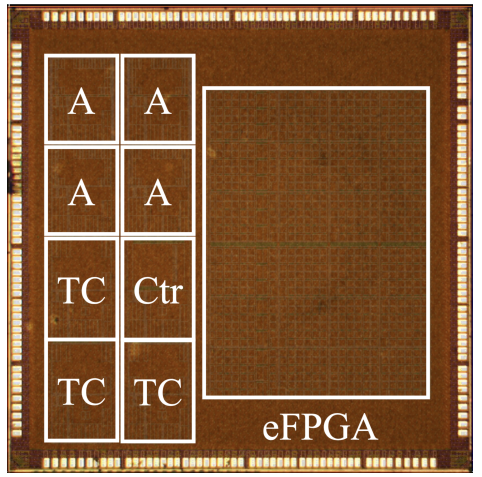
BRGTC1
IBM 130nm
2x2mm



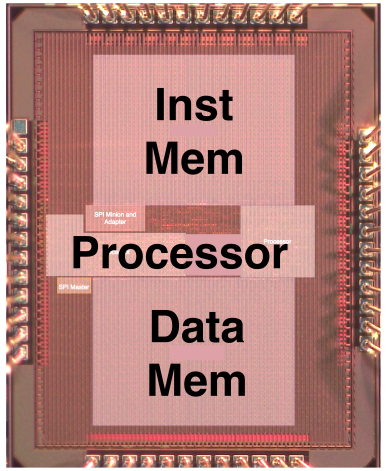
Celerity
TSMC 16nm
5x5mm



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

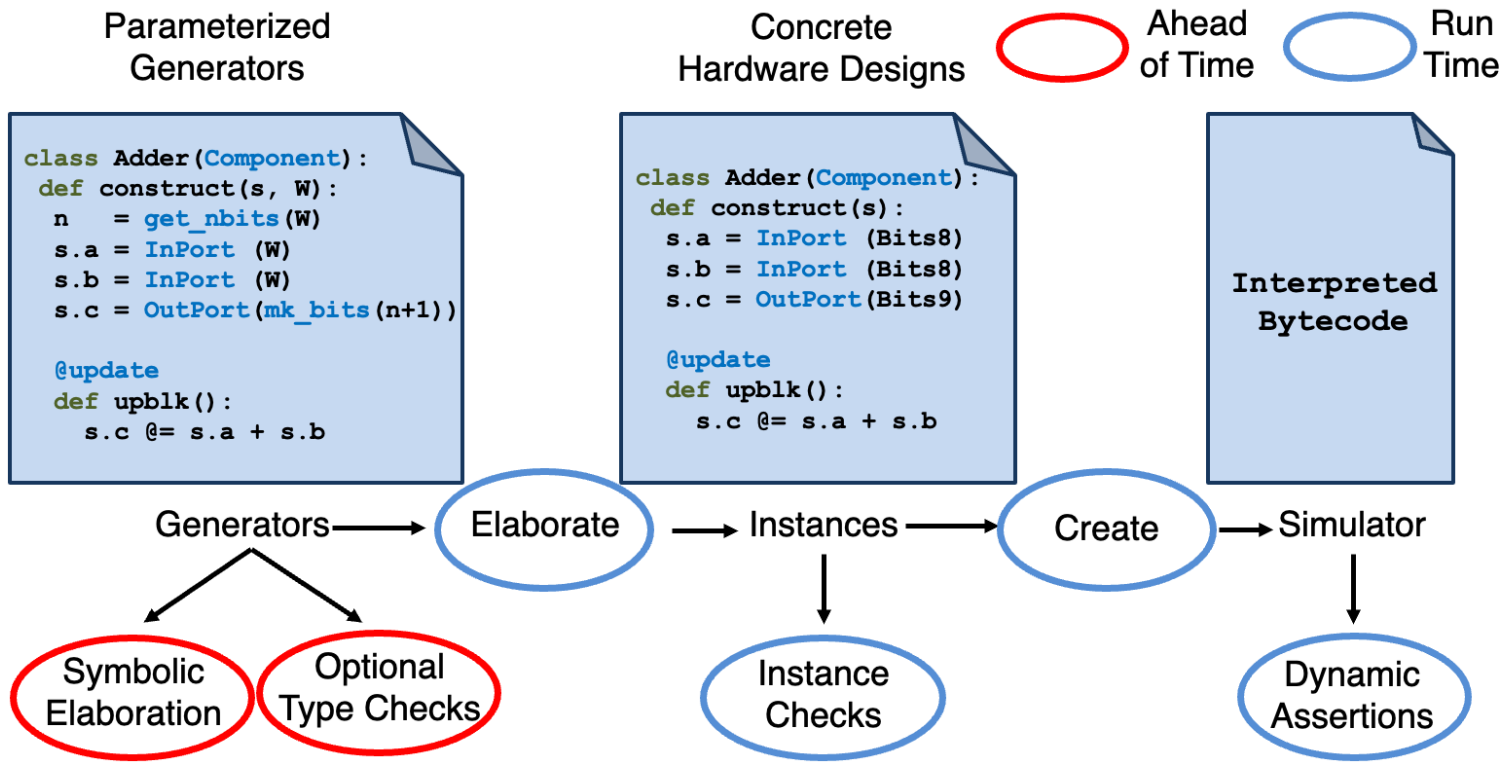
PyMTL3 Introduction

Target Generator Properties

Optional Type Checking

Symbolic Elaboration

Evaluation



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*

N/M Connector Example

```
# Declare Component
class NMConn(Component):
 def construct(s, N, M):
  s.a = InPort (N)
  s.b = OutPort(M)

  # Only valid if N == M
  connect(s.a, s.b)
```

```
# 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
```

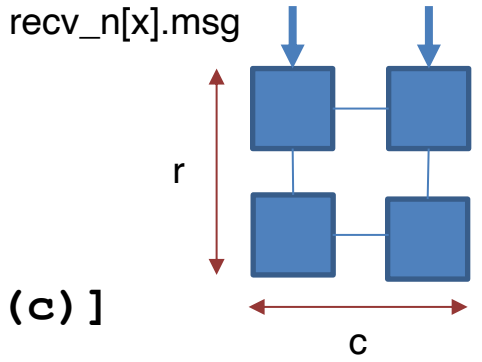
Target Property #2: Bounded Array Indexing

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




```
# 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!
```

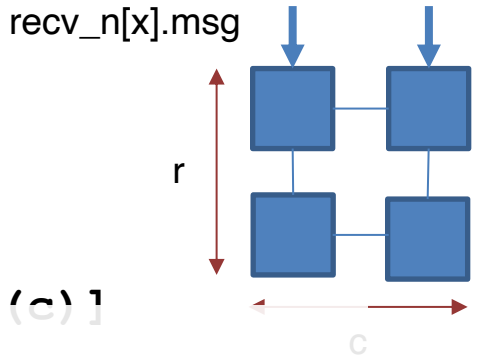
Target Property #2: Bounded Array Indexing

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*

Example bug caught when connecting with square reconfigurable array

Reconfigurable Array Example




```
# 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:
     # Connect bottom row of PEs to c
     connect(s.pes[y*r+x].recv[N].msg, s.recv_n[x].msg)
```

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

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

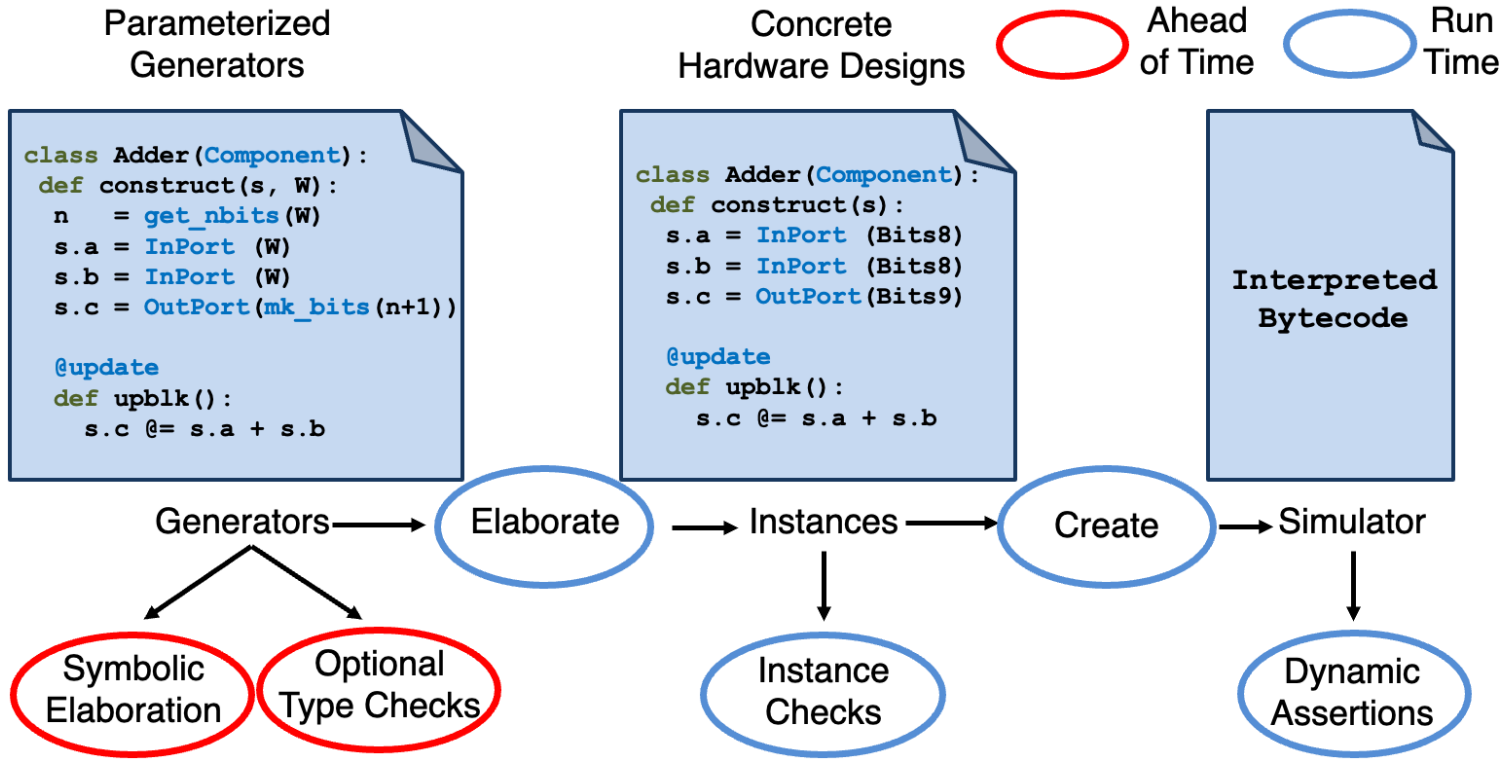
PyMTL3 Framework

Target Generator Properties

Optional Type Checking

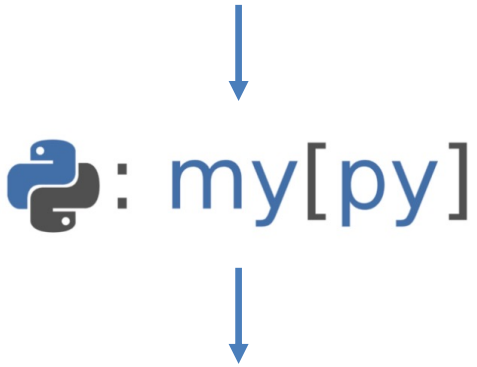
Symbolic Elaboration

Evaluation



Mypy: An Optional Type Checker for Python

Type-Annotated Python Programs (.py)



Type check or not

```
def f(p:int)->bool:
  if p > 0:
    return True
  else:
    return False
```

Success: all types are known ahead of time based on annotation and inference

```
def f(p:str)->bool:
  if p > 0:
    return True
  else:
    return False
```

Fail: cannot compare a string to an integer

```
def f(p)->bool:
  # p: Any
  if p > 0:
    return True
  else:
    return False
```

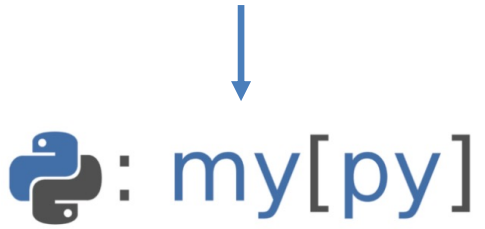
Success: type of p not known ahead of time (Any); be permissive and allow operations on p

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

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

Generator (.py)




```
# Hardware Data Types
class Bits: ...
class Bits1(Bits): ...

# Hardware Signal Types
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
T_ConnectN = TypeVar("T_ConnectN", bound=Bits)
def connect(l: Signal[T_ConnectN], r: Signal[T_ConnectN]) -> None: .
```

Repurposing Mypy for Hardware: Full-Adder Example

```
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 ***parametrized*** bitwidths and ***bitwidth-mutating*** operations
- Very challenging to encode ***array indices*** (and also ***port directions***, ***hierarchical references)***
- In addition, Mypy does not account for ***path conditions*** (using if-conditions in generators)

Adder Generator in PyMTL3

```
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: Checking Generator Properties in Dynamic Hardware Description Languages

Motivation

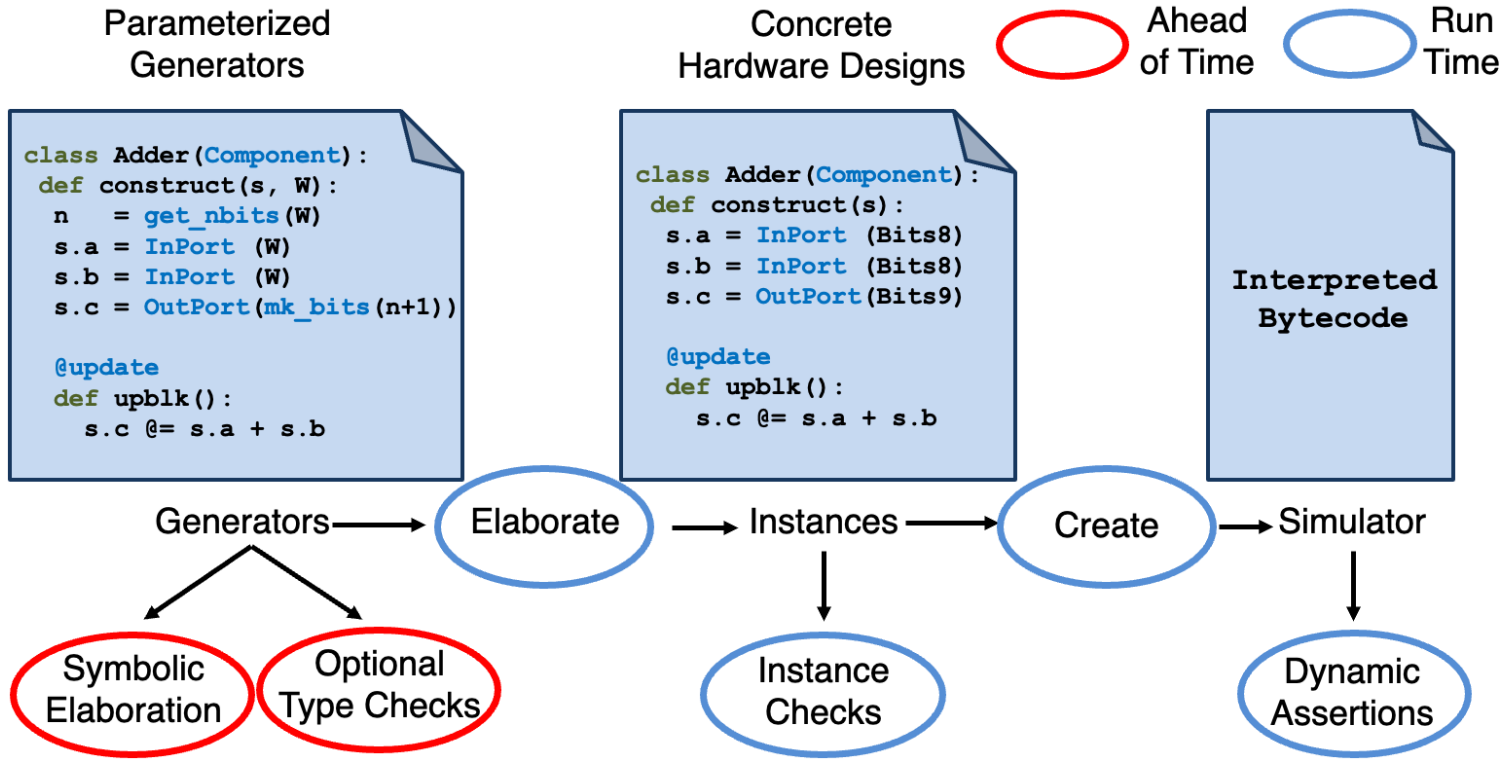
PyMTL3 Framework

Target Generator Properties

Optional Type Checking

Symbolic Elaboration

Evaluation



Symbolic Elaboration Motivation

```
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 )
```

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 != 1+n) for all positive integer n?

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		
Name	Type & Metadata	Definition Condition

Adder Abstract Generator Model		
Name	Type	Definition Condition

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

Name	Type & Metadata	Definition Condition
W	**Bits; generator arg**	**true**

Adder Abstract Generator Model

Name	Type	Definition Condition

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

Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	**int; get_nbits(W)**	**true**

Adder Abstract Generator Model

Name	Type	Definition Condition

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

Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	int; get_nbits(W)	true

Adder Abstract Generator Model

Name	Type	Definition Condition
a	**InPort[W]**	**true**
b	**InPort[W]**	**true**

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

Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	int; get_nbits(W)	true

Adder Abstract Generator Model

Name	Type	Definition Condition
a	InPort[W]	true
b	InPort[W]	true
out	**OutPort[n+1]**	**true**
carry	**Wire[n+1]**	**true**

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

Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	int; get_nbits(W)	true

Adder Abstract Generator Model

Name	Type	Definition Condition
a	InPort[W]	true
b	InPort[W]	true
out	OutPort[n+1]	true
carry	Wire[n+1]	true
sum	**Wire[W]**	**true**

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

Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	int; get_nbits(W)	true

Adder Abstract Generator Model

Name	Type	Definition Condition
a	InPort[W]	true
b	InPort[W]	true
out	OutPort[n+1]	true
carry	Wire[n+1]	true
sum	Wire[W]	true
fa	**List[FullAdder] of n**	**true**

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		
Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	int; get_nbits(W)	true
i	**int; 0 <= i < n**	**true**

Adder Abstract Generator Model		
Name	Type	Definition Condition
a	InPort[W]	true
b	InPort[W]	true
out	OutPort[n+1]	true
carry	Wire[n+1]	true
sum	Wire[W]	true
fa	List[FullAdder] of n	true

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:

- ¬ (n + 1 = 1 + W) ∧ true ∧ true ∧ (n = W)
  - true ∧ true: from use condition and definition condition

SMT solver finds the constraint unsatisfiable => proof!

Adder Symbol Table

Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	int; get_nbits(W)	true
i	int; 0 <= i < n	true

Adder Abstract Generator Model

Name	Type	Definition Condition
a	InPort[W]	true
b	InPort[W]	true
out	OutPort[n+1]	true
carry	Wire[n+1]	true
sum	Wire[W]	true
fa	List[FullAdder] of n	true

Symbolic Elaboration with an Adder Generator Example

Property: bounded array indexing

```
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

Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	int; get_nbits(W)	true
i	int; 0 <= i < n	true

Adder Abstract Generator Model

Name	Type	Definition Condition
a	InPort[W]	true
b	InPort[W]	true
out	OutPort[n+1]	true
carry	Wire[n+1]	true
sum	Wire[W]	true
fa	List[FullAdder] of n	true

Symbolic Elaboration with an Adder Generator Example

Property: bounded array indexing

```
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

Name	Type & Metadata	Definition Condition
W	Bits; generator arg	true
n	int; get_nbits(W)	true
i	int; 0 <= i < n	true

Adder Abstract Component Model

Name	Type	Definition Condition
a	InPort[W]	true
b	InPort[W]	true
out	OutPort[n+1]	true
carry	Wire[n+1]	true
sum	Wire[W]	true
fa	List[FullAdder] of n	true

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

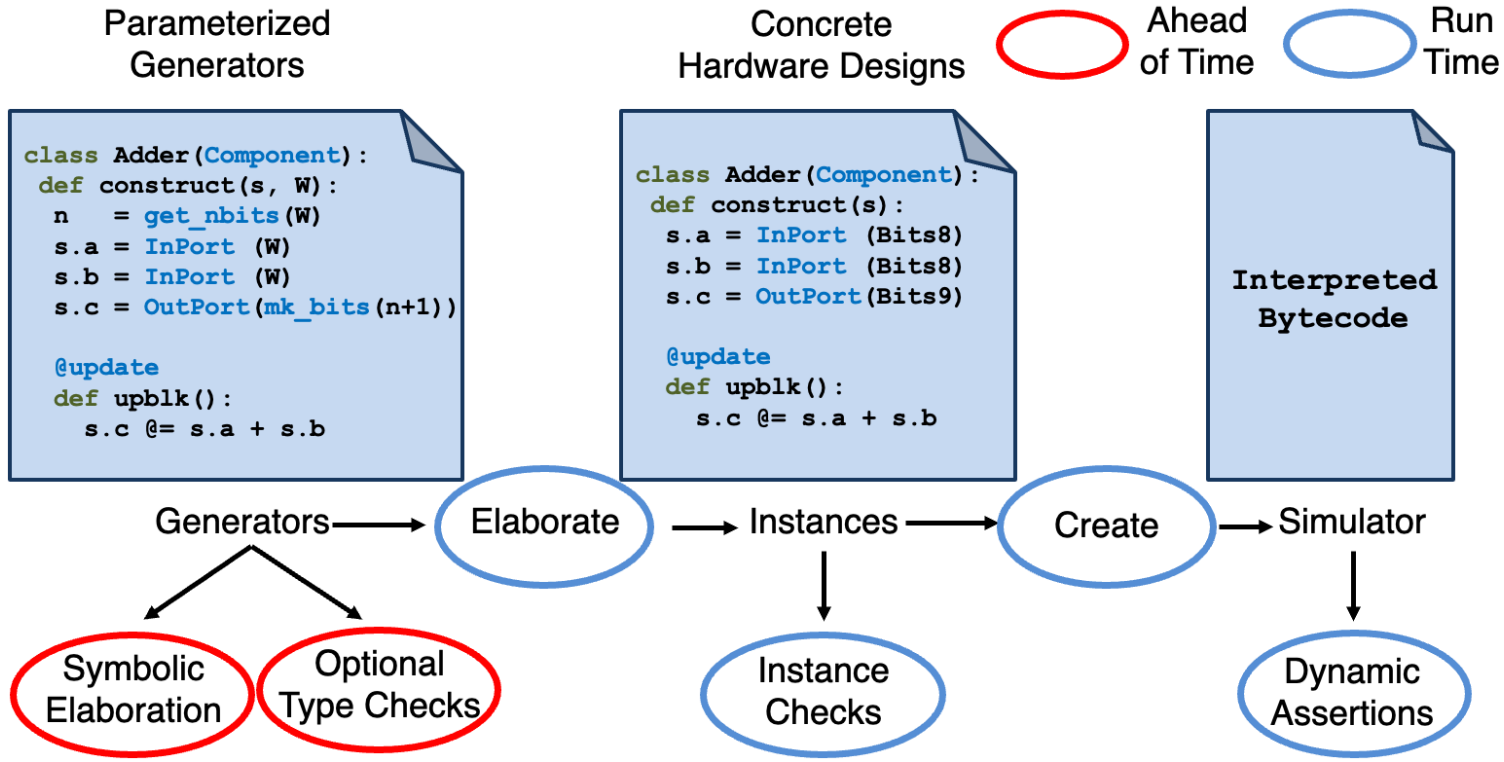
PyMTL3 Framework

Target Generator Properties

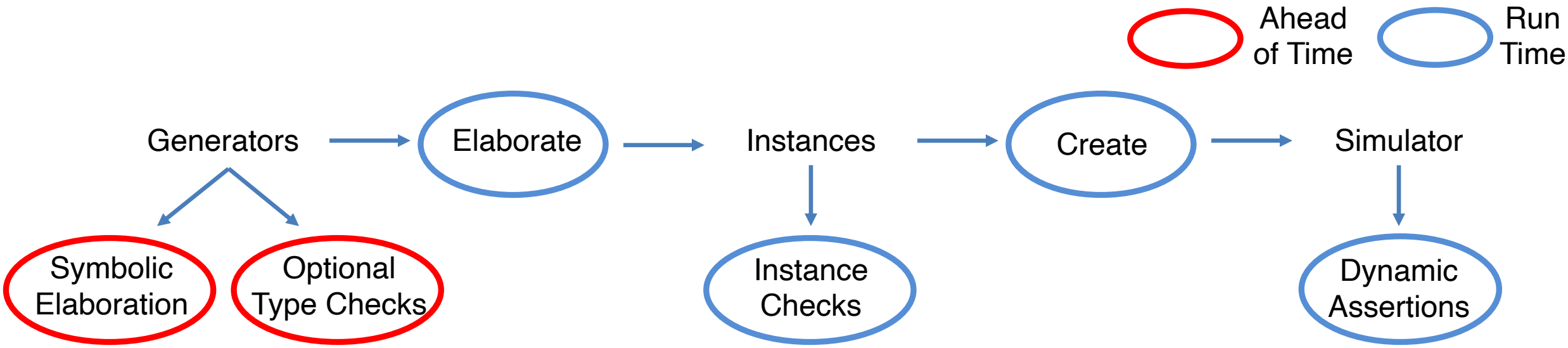
Optional Type Checking

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

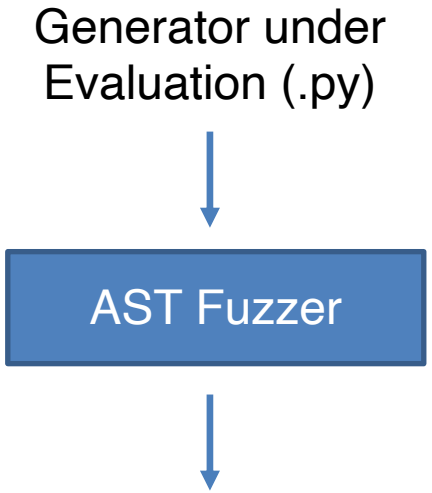
PyMTL3 + Symbolic Elaboration (S)

- Symbolic elaboration on generators ahead of time
- Still perform all instance checks and dynamic assertions at run time

Evaluation Methodology: Generators used in Experiments

	Generator	Generator Lines of Code	Instance Lines of Code	Instance
Standard Library IPs	LFSR	31	23	32-bit register
	Gray Encoder/Decoder	42	76	32-bit input
	Priority Encoder	45	79	32-bit input
	Round-Robin Arbiter	49	88	4 requesters
	FIFO	125	174	32-bit 2-element
Stand-alone Designs	Divider	172	535	32-bit data path
	Processor	903	2135	Single-core RV32-IM
	CGRA	1170	4004	8x8 32-bit PE array

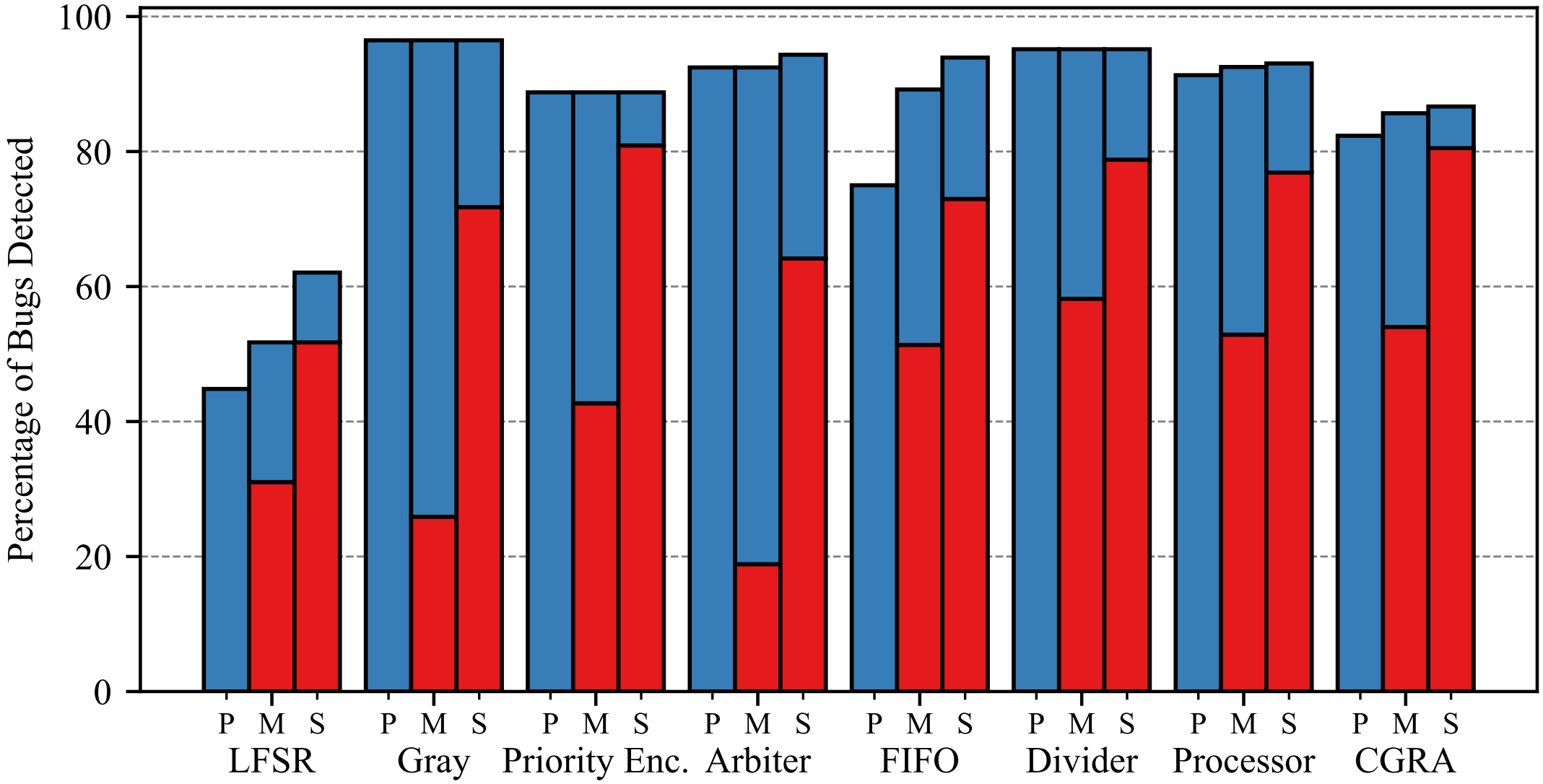
Evaluation Methodology: AST Fuzzer



AST fuzzer may inject benign mutations – a mutated generator is not always bugged!

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

Evaluation: Bug Detection Results



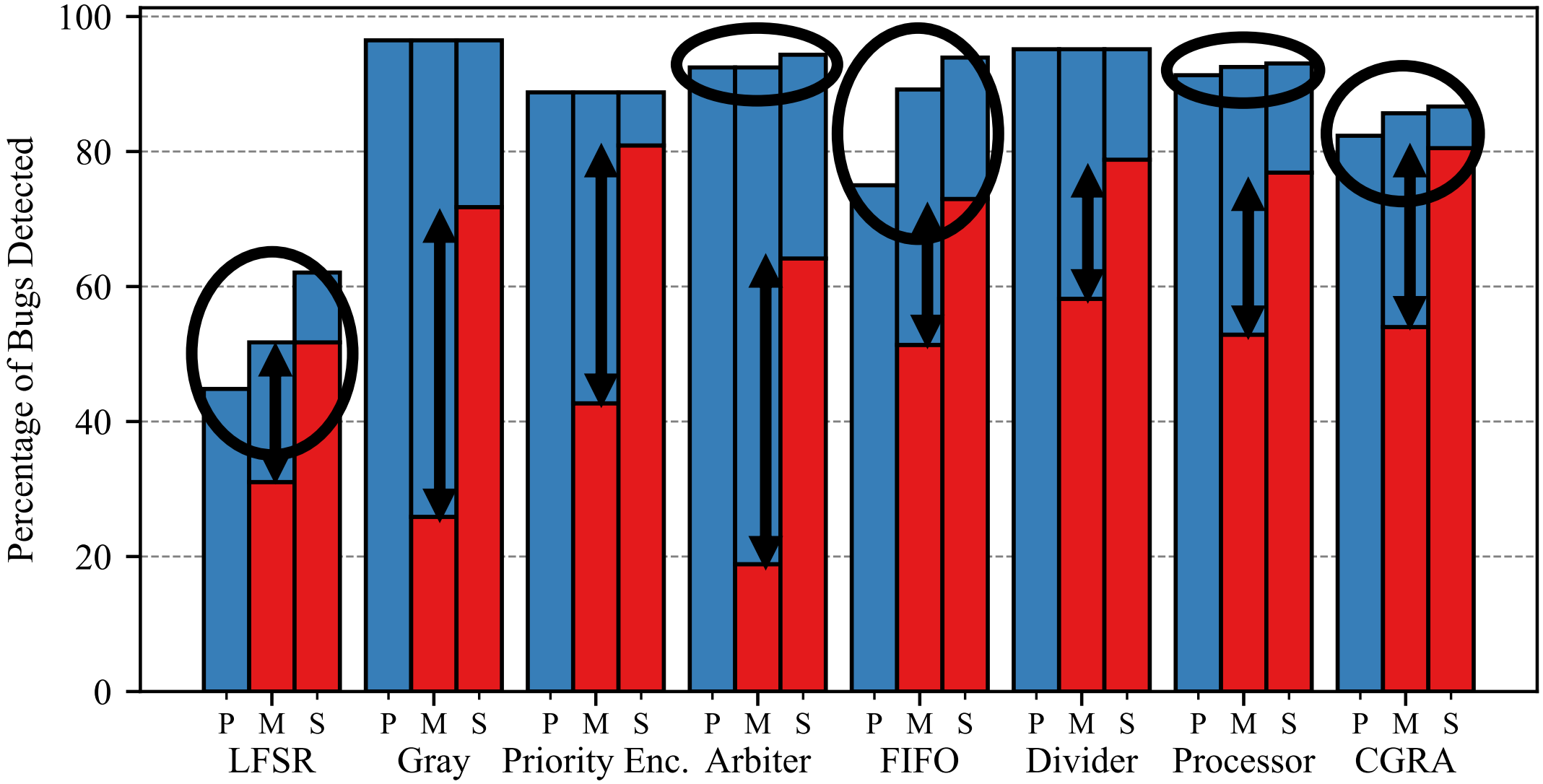
Ahead of Time
Run Time

P: PyMTL3

M: PyMTL3 with MyPy

S: PyMTL3 with Symbolic Elaboration

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: Checking Generator Properties in Dynamic Hardware Description Languages

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.