Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages

Peitian Pan, Shunning Jiang, Yanghui Ou, Christopher Batten
Cornell University
Ithaca, NY, USA
{pp482,sj634,yo96,cbatten}@cornell.edu

ABSTRACT

Recent research in hardware development methodologies has argued for using programmatic hardware generators and dynamically typed behavioral models to significantly improve hardware design and verification productivity. Dynamic hardware description languages (HDLs) promise to realize these productivity benefits by composing generated hardware instances with models backed by virtually any dynamically typed code. However, dynamic HDLs generally lack powerful static checking capabilities which prevents these HDLs from detecting generator bugs early in the design-debug cycle. In this paper, we propose symbolic elaboration, a novel static checking technique, to provide useful static correctness guarantees for hardware generators in dynamic HDLs. Symbolic elaboration builds an abstract representation for a given hardware generator and translates its properties into constraints solvable by a satisfiability modulo theory (SMT) solver. Symbolic elaboration provides static correctness guarantees for generator properties including matching bitwidths, correct local port directions, bounded array indexing, and valid hierarchical references. We evaluate symbolic elaboration on eight hardware generators in a state-of-the-art dynamic HDL. Our evaluation shows that, on average, symbolic elaboration can statically detect 90.6% of randomly injected bugs and 53.0% more bugs than an off-the-shelf static type checker.

CCS CONCEPTS

• **Hardware → Hardware description languages and compilation**.

KEYWORDS

hardware description languages, hardware generators, static analysis

ACM Reference Format:
Peitian Pan, Shunning Jiang, Yanghui Ou, Christopher Batten. 2023. Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages . In *21st ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '23), September 21–22, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3610579.3611076

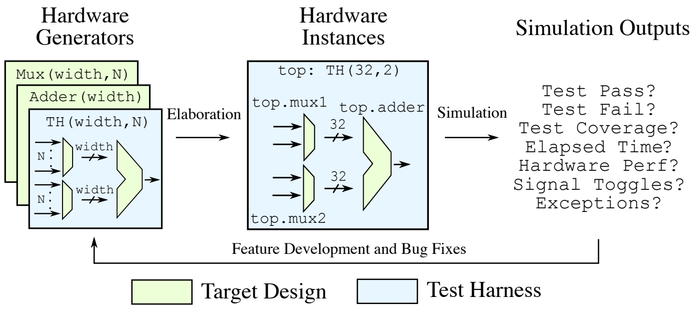


Figure 1: A Typical Design-Debug Cycle with Hardware Generators.

1 INTRODUCTION

The slowdown of Moore's law and the breakdown of Dennard scaling have driven computer architects towards more specialized hardware designs to meet applications' growing performance and energy efficiency demands. However, such specialized hardware designs tend to have high non-recurring engineering (NRE) costs that hinder the research and development of promising hardware systems. Recent research in hardware development methodologies addresses the high NRE costs of specialized hardware designs in two ways: (1) the promotion of programmatic hardware generators to maximize design reuse [3, 5, 13, 15, 23, 29, 32] and (2) the extensive use of dynamically typed behavioral models to facilitate the creation of highly parametrized and polymorphic test harnesses, golden reference models, and cycle-approximate hardware models [11, 14, 15, 17, 19, 22]. Researchers have embedded hardware description languages (HDLs) into dynamic languages like Python to realize the benefits of these two proposals [9, 10, 15, 19, 20]. These *dynamic HDLs* increase hardware design and verification productivity by allowing sophisticated hardware generators and composition of generated hardware instances with behavioral models backed by virtually any dynamically typed code.

Unfortunately, dynamic HDLs generally do not provide any static correctness guarantees for hardware generators and defer correctness checks to either elaboration or simulation. Figure 1 illustrates a typical three-step hardware design cycle in existing HDLs. The early step of the design cycle is to create or reuse hardware generators for the target design and its test harness. The middle step performs *elaboration*, which takes the top-level generator (typically a test harness generator) and a set of parameters to construct a hierarchy of hardware instances. The final step of the design cycle performs simulation on the elaborated hierarchy of instances and generates simulation outputs for feature development and bug fixing purposes. As an example, dynamic HDLs perform virtually no checks on hardware generators `Mux`, `Adder`, and `TH` in Figure 1 because the

Table 1: Existing HDLs and Their Characteristics

HDLs	Programmatic Generators	Dyn. Typed Behavioral Models	Num. of Properties Enforced			Target of Properties Enforcement		
			AoT	ET	ST	AoT	ET	ST
Traditional Static HDLs Verilog/SystemVerilog, VHDL			○	●	○		Single Instance	
High-Level Static HDLs Bluespec SystemVerilog, Cλash	✓		●	◒	○	All Instances	Single Instance	
Constructional Static HDLs Chisel, SpinalHDL, Lava	✓		◒	●	○	All Instances	Single Instance	
Dynamic HDLs PyRTL, Migen, MyHDL, PyMTL3	✓	✓	○	●	●		Single Instance	Single Input
Dynamic HDLs with Symbolic Elaboration*	✓	✓	●	◒	◒	All Instances	Single Instance	Single Input

Programmatic generators: hardware generators that programmatically generate hardware instances. HW: hardware. Hardware properties can be enforced at three times: ahead of time (AoT) by checking hardware generators (static); elaboration time (ET) by checking hardware instances; simulation time (ST) by checking signal assignments in simulation. We focus on hardware properties discussed in §2.2 in this paper. All Instances: the given hardware property is guaranteed to hold on all instances of the target generator under all input; Single Instance: the given hardware property is guaranteed to hold on the target instance under all input; Single Input: the given hardware property is guaranteed to hold on the target instance under the given input. ●/◒/○ : almost all/some/no properties enforced; Dyn. typed: dynamically typed. *: our proposal statically provides strong generator correctness guarantees (All Instances) for dynamic HDLs.

concrete generator parameters are not available before elaboration. Given a set of concrete generator parameters (e.g., 32-bit data path width for adder and muxes, 2-input muxes), dynamic HDLs elaborate generators into a hierarchy of hardware *instances* and checks for structural connection errors during elaboration among those instances (e.g., dynamic HDLs can verify the signals connected to the inputs of the adder have the same bitwidth 32). And further given a set of concrete test vectors, dynamic HDLs simulate the target design with the test harness and check for behavioral errors. The lack of capable static checking abilities creates a long design-debug cycle where design issues can only be identified and fixed after elaboration or simulation, which hinders design productivity.

In this paper, we propose *symbolic elaboration* (SE) to provide static correctness guarantees for hardware generators in dynamic HDLs and shorten the design-debug cycle. We first explore existing off-the-shelf Python static type checkers for generators and observe that they are too specific to software programs and cannot effectively verify critical hardware generator properties. To overcome the above limitations, we observe that most hardware generators can be expressed as an abstract model whose structural and behavioral models rely on a *symbolic* generator parameter rather than concrete values. We propose symbolic elaboration, a novel technique that translates these abstract structural and behavioral models into integer constraints that can be solved by SMT solvers. The proof of the correctness of hardware generators is obtained if the solver finds such constraints unsatisfiable, and a counterexample of the violation of generator properties is generated if the solver finds a satisfiable assignment of variables. Our prototype implementation of symbolic elaboration can verify critical generator properties including matching bitwidths, correct local port directions, bounded array indexing, and valid hierarchical references.

This paper makes the following contributions:

- We apply an off-the-shelf static type checker to hardware generators in dynamic HDLs and analyze its limitations (§3).
- We propose symbolic elaboration to overcome the limitations of off-the-shelf static type checkers and statically verify critical hardware generator properties (§4).
- We evaluate a prototype SE implementation on eight RTL design generators in a state-of-the-art dynamic HDL using a mutation-based abstract syntax tree (AST) fuzzer (§5).

This paper is organized as follows: §2 provides the background for existing HDLs and the target hardware generator properties we focus on in this paper. §3 describes how Mypy, an existing static type checkers for Python, can be applied to check hardware generators and its limitations. §4 discusses symbolic elaboration which overcomes Mypy's limitations. §5 presents the evaluation of a prototype SE implementation. §6 discusses related works.

2 BACKGROUND

We begin by providing some background on hardware description languages and the target hardware generator properties we focus on in this paper. §2.1 introduces existing HDLs. §2.2 introduces the hardware properties we try to verify in this paper.

2.1 Existing HDLs

Table 1 summarizes existing HDLs based on the following characteristics: generator support, dynamically typed behavioral model support, when hardware properties are enforced in the hardware design cycle, and the target on which the given properties are enforced. Table 1 shows that (1) traditional HDLs like SystemVerilog and VHDL fail to support programmatic generators and dynamically typed behavioral modules; (2) high-level HDLs and constructional HDLs are able to enforce hardware properties on all instances of some generators but generally do not support dynamically typed behavioral models; (3) dynamic HDLs support both programmatic generators and dynamically typed behavioral modules but do not enforce hardware properties ahead of time.

Traditional Static HDLs – Verilog/SystemVerilog [2] and VHDL [1] belong to traditional statically typed HDLs. Verilog, SystemVerilog, and VHDL all support limited forms of hardware generation through generate statements (if- and for-statements). However, hardware generators in these HDLs cannot leverage more advanced programmatic constructs such as recursive functions, object-oriented abstractions, and advanced data structures beyond lists. This makes it challenging to programmatically construct hardware instances with these HDLs. Their static type systems also impose challenges on creating dynamically typed components. Traditional HDLs enforce hardware properties by type checking the elaborated hierarchy of hardware instances, which happens in the middle of a hardware design cycle.

High-Level Static HDLs – Bluespec SystemVerilog [24] has a powerful static type system. It supports programmatic hardware generators and can type check the generators to discover potential design issues early in the design cycle. However, Bluespec cannot verify critical properties such as bitwidth mismatches in vector slicing operations and defers this check to elaboration. Cλash [4] is a Haskell dialect for hardware development. It benefits from Haskell's static type system and is able to type check the generators before elaboration. Both Bluespec SystemVerilog and Cλash adopt a different level of abstraction from the conventional register-transfer level abstraction and do not support dynamically typed components.

Constructional Static HDLs – Chisel [5] and SpinalHDL [30] are HDLs embedded in Scala. Both Chisel and SpinalHDL support programmatic hardware generators. Lava [7] is an HDL embedded in Haskell capable of programmatic generators. Unlike Cλash, Lava does not leverage Haskell's type system to type check hardware generators. Constructional static HDLs mainly enforce HDL hardware invariants by type checking the elaborated hardware instances. More specifically, for Chisel, type checks on hardware instances happen through a detailed analysis on FIRRTL, a post-elaboration intermediate representation of hardware [13]. Constructional static HDLs focus on hardware construction using various modeling primitives embedded in the statically typed host language (such as Scala) and do not support dynamically typed components.

Dynamic HDLs – Dynamic HDLs support both programmatic hardware generators and dynamically typed behavioral models thanks to the flexibility of its host language. PyRTL [9], Migen [20], MyHDL [10], PyMTL [19], and PyMTL3 [15] are all HDLs embedded in Python, a dynamically typed programming language. Figure 2 shows a parametrized adder generator and a dynamically typed polymorphic test harness in PyMTL3, a state-of-the-art dynamic HDL. The polymorphic test harness highlights how dynamic HDLs help improve verification productivity by promoting the reuse of parametrizable verification modules. In the polymorphic test harness example, the input setter and output checker of the test harness can be abstracted as functions taking variable number of arguments. Therefore, it is possible to reuse the harness and simulation setup across different designs by passing in different functions. Almost all existing dynamic HDLs do not check generators due to the lack of static checking capabilities. Instead, most dynamic HDLs rely on elaboration- and simulation-time checks on a flattened module hierarchy to enforce critical hardware properties.

```
class FullAdder(Component):
  def construct(s):
    s.a   = InPort(Bits1);  s.b    = InPort(Bits1)
    s.cin = InPort(Bits1)
    s.sum = OutPort(Bits1); s.cout = OutPort(Bits1)

    @update
    def upblk():
      s.sum  @= s.cin ^ s.a ^ s.b
      s.cout @= ((s.a ^ s.b) & s.cin) | (s.a & s.b)

class Adder(Component):
  def construct(s, Width):
    n = get_nbits(Width)
    s.a   = InPort(Width);
    s.b = InPort(Width)
    s.out = OutPort(mk_bits(n+1))

    s.fa  = [FullAdder() for _ in range(n)]
    s.carry = Wire(mk_bits(n+1));
    s.sum = Wire(Width)

    s.carry[0] //= 0
    for i in range(n):
      s.fa[i].a    //= s.a[i]
      s.fa[i].b    //= s.b[i]
      s.fa[i].cin  //= s.carry[i]
      s.carry[i+1] //= s.fa[i].cout
      s.sum[i]     //= s.fa[i].sum

    s.out //= lambda: concat(s.carry[n], s.sum)
```

(a) PyMTL3 Adder. `Adder` is parametrized by its data path width (`Width`); it uses the `FullAdder` module as its sub-components; ports and wires are constructed with their bitwidth in parentheses; `//=` is the connection operator that connects two ports or wires together.

```
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   Adder(),    [
14     (  3,     5,    8),
15     (  1,    42,   32),
16     ( 10,     8,   18),
17     (  0,     0,    0),
18   ],
19   lambda m,t: (Assign(m.a,t[0]), Assign(m.b,t[1])),
20   lambda m,t: Eq(m.out,t[2])
21 )
```

(b) Polymorphic Test Harness. Line 12-21 show how to specialize the test harness for the adder, which includes test vectors and input setting/output checking lambda functions; the harness is dynamically typed to allow passing in customized input and output functions of any valid Python code.

Figure 2: Adder in PyMTL3, a State-of-the-Art Dynamic HDL.

Scope of This Paper – We focus on symbolic elaboration in the context of dynamic HDLs in this paper. Symbolic elaboration is particularly well suited to dynamic HDLs because it helps fill the static checking gap commonly found in these languages. We also note that the core idea of symbolic elaboration can be applied to other HDLs to enable more powerful static checking on generators than currently available in static HDLs.

2.2 Target Hardware Generator Properties

In this paper, we focus on four key hardware generator properties on design correctness and synthesizability.

Matching Bitwidths – One important aspect of structural modeling is modeling the circuit using a set of interconnected signals. Therefore, it is crucial for HDLs to verify that structurally connected signals have the same bitwidth. The matching bitwidth property also applies to behavioral modeling, where the bitwidths of operands in arithmetic operations should be equal. As an example, the left-hand side (LHS) and right-hand side (RHS) of connections on line 25-29 in Figure 2 (a) should have the same bitwidth.

Correct Local Port Directions – Interconnected signals in circuits are generally further elaborated into *nets* of signals where at most one active signal drives all other signals in the same net. A complete net representation requires extensive cross-module analysis on the elaborated component [13, 15] and may not be possible using static analysis. *Local* port direction analysis focuses on the connections within a component and can find common direction issues such as attempting to drive input-only ports from inside a component. As an example, only an output port can be used on the LHS of a signal assignment (line 31 in Figure 2 (a)).

Bounded Array Indexing – Hardware generators often leverage arrays to model signals and sub-components. Design issues can therefore arise from out-of-bound array indexing. Unlike array index calculation in programs which can have arbitrary computation, array index generation in hardware generators generally consists only of simple arithmetics over constants, generator parameters, and loop induction variables. Therefore, static analysis on hardware generators should be able to detect almost all out-of-bound array indexing. As an example, array indices have to be smaller than the array length (line 23, 25-29 in Figure 2 (a)).

Valid Hierarchical References – Traditional HDLs like Verilog allow accessing any attribute in the elaborated module hierarchy using a globally unique *hierarchical name* to facilitate hardware testing [2]. However, arbitrary hierarchical references do not model actual hardware behaviors. Synthesizable hardware generators must communicate through input and output data ports between two immediate levels in the module hierarchy. As a concrete example, the valid hierarchical reference property requires that only the input and output ports of the `FullAdder` sub-components can be accessed (line 25-29 in Figure 2 (a)).

3 CHECKING GENERATOR PROPERTIES WITH AN OFF-THE-SHELF STATIC TYPE CHECKER

In this section, we discuss how to apply Mypy, an off-the-shelf static type checkers for Python [18], to statically check PyMTL3 hardware generators. We present type annotations for the PyMTL3 hardware modeling domain-specific language (DSL), which Mypy requires to analyze hardware generators. We show that Mypy can verify simple generator properties and discuss its limitations.

3.1 Type Annotations for PyMTL3 DSL and Generators

As a static type checker, Mypy leverages Python's type annotation syntax to type check programs. Two kinds of type annotations are necessary to repurpose Mypy for hardware generator properties: annotations for the PyMTL3 hardware modeling DSL (Figure 3 (a)) and annotations for the target generator (Figure 3 (b)).

```python
# Hardware data types
class Bits: ...
class Bits1(Bits): ...
class Bits32(Bits): ...

def mk_bits(nbits: int) -> Type[Bits]: ...
def get_nbits(Width: Type[Bits]) -> int: ...

# Hardware signal types
T_Sig = TypeVar("T_Sig", bound=Bits)
class Signal(Generic[T_Sig]):
  def __init__(s, Width: Type[T_Sig]) -> None:
    ...
  def __xor__(s, o: Signal[T_Sig]) -> Signal[T_Sig]:
    ...
  def __and__(s, o: Signal[T_Sig]) -> Signal[T_Sig]:
    ...
  def __or__(s, o: Signal[T_Sig]) -> Signal[T_Sig]:
    ...

InPort = OutPort = Wire = Signal

# Hardware modeling primitive types
T_Con = TypeVar("T_Con", bound=Bits)
@overload
def connect(l: Signal[T_Con],
            r: Signal[T_Con]) -> None: ...
@overload
def connect(l: Signal[T_Con],
            r: int) -> None: ...

def concat(*args: List[Any]) -> Signal[Bits]: ...
```

(a) Type-Annotated Core PyMTL3 Hardware Modeling DSL

```python
class FullAdder(Component):
  def __init__(s) -> None: ...
  def construct(s) -> None:
    # All ports have type Signal[Bits1]
    s.a    = InPort(Bits1);  s.b    = InPort(Bits1)
    s.cin  = InPort(Bits1)
    s.sum  = OutPort(Bits1); s.cout = OutPort(Bits1)

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

T_Adder = TypeVar("T_Adder", bound=Bits)
class Adder(Component, Generic[T_Adder]):
  def __init__(s, Width: Type[T_Adder]) -> None:
    ...
  def construct(s, Width: Type[T_Adder]) -> None:
    n       = get_nbits(Width)
    s.a     = InPort(Width)          # Signal[T_Adder]
    s.b     = InPort(Width)          # Signal[T_Adder]
    s.out   = OutPort(mk_bits(n+1)) # Signal[Bits]
    s.fa    = [FullAdder() for _ in range(n)]
                                     # List[FullAdder]
    s.carry = Wire(mk_bits(n+1))     # Signal[Bits]
    s.sum   = Wire(Width)            # Signal[T_Adder]

    for i in range(n):
      if i >= 0:
        # Both sides of connect are of Signal[Bits1]
        connect(s.fa[i].a   , s.a[i]      )
        connect(s.fa[i].b   , s.b[i]      )
        connect(s.fa[i].cin , s.carry[i]  )
        connect(s.carry[i+1], s.fa[i].cout)
        connect(s.sum[i]    , s.fa[i].sum )
      if i == 0:
        # Left: Signal[Bits1]; right: int
        connect(s.carry[i], 0)

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

(b) Type-Annotated Adder Generator Checked by Mypy

Figure 3: Type-Annotated PyMTL3 Generators

Hardware Data Types – Hardware data types in PyMTL3 are Python classes that represent a specific bitwidth and are used to specify the bitwidth of signals. PyMTL3 dynamically generates and caches such class objects during import, which is a dynamic behavior that cannot be type-annotated precisely. Instead, line 2-5 in Figure 3 (a) exhaustively lists *all* data types used in the generators (only `Bits1` and `Bits32` is used in this example). When the type checker cannot determine the exact bitwidth of a signal, the base class object `Bits` is used. PyMTL3 also provides functions that converts an integer to and from a bitwidth type (i.e., a *class* object), which are annotated on line 6-7. The annotation again is imprecise because the exact bitwidth is not known at annotation time.

Hardware Signal Types – Line 10-21 annotate the signal type in the PyMTL3 modeling DSL, which is used whenever a port (input or output) or wire is instantiated. We declare `Signal` as a generic type over type variable `T_Sig`, which represents a hardware data type indicating the signal's bitwidth (achieved with the `bound` argument of the type variable). Line 12 in Figure 3 (a) declares that only class objects are allowed to be passed into signal constructors, which enables the detection of misuse of other invalid objects to construct a signal during type check. Line 14-19 specify the type signature of common bitwise operators: both sides of the operation should be a signal of the same bitwidth, and the operation returns a signal of the same bitwidth. This allows Mypy to detect simple bitwidth mismatch errors in generators. It is also worth noting that encoding the direction of signals into Mypy's type system is challenging. Therefore, we disregard the direction of signals while annotating the PyMTL3 modeling DSL (line 21 in Figure 3 (a)).

Hardware Modeling Primitive Types – Line 24-30 show the type annotations of the `connect` method, which is used to connect two signals in a design. The signature of `connect` is overloaded to support connecting signals to other signals (line 24-27) and integers (line 28-30). The method signature ensures that if two signals are connected, they must have the same bitwidth to satisfy the matching bitwidth property. If a signal is connected to an integer, no checks are necessary because the PyMTL3 semantics ensures that the integer will be cast to fit the bitwidth of the other signal. Finally, line 32 shows the signature of the `concat` method, which is used to concatenate a variable number of explicitly sized signals into one. Since it is not always possible to know the exact bitwidth of the resulting signal, `Signal[Bits]` is an imprecise annotation we can do for `concat`'s return type.

Hardware Generator Annotations – Figure 3 (b) shows the type-annotated adder generator. Unlike annotations for the PyMTL3 hardware modeling DSL which can be built into the dynamic HDL, hardware designers need to manually annotate their generator design to get it type checked by a static type checker. Fortunately, only a few annotations in the generator class or function definition are required. On line 2, 3, and 10, the `->None` symbol is a dummy annotation that enables Mypy checking at the function scope. For generators with parameters, the type checker requires annotations that declare the generator class as having generic type (shown on line 14-15) over the hardware data types. Mypy can infer the types of signals based on its instantiation (line 20-22).

Putting It All Together – with annotations for the PyMTL3 modeling DSL and the generators, Mypy can verify the matching bitwidths for the bit-wise operations on line 11-12 in Figure 3 (b) because all operands are explicitly declared to be single-bit wide (`Bits1`). Mypy can also verify the matching bitwidths for structural connections on line 31-35 because these signals are explicitly sized.

3.2 Limitations of Mypy

Despite Mypy's success in verifying parts of the bitwidth matching property, it has several limitations.

Mypy Cannot Statically Verify All Matching Bitwidths – In the adder generator in Figure 3 (b), Mypy cannot reason about the exact bitwidth of `s.out` (line 22) and the result of `concat()` (line 43). Therefore, it fails to verify that the LHS and RHS of line 43 in Figure 3 (b) have the same bitwidth. We make the observation that *Mypy is only able to verify matching bitwidths that are not parametrized.* Unfortunately, many hardware generators rely on parametrized signal bitwidths (line 20-21) and deriving new hardware data types (line 22,25), which undermine Mypy's effectiveness.

Verifying Other Generator Properties with Mypy is Challenging – Besides the matching bitwidth property, it is challenging to encode other generator properties discussed in §2.2 into Mypy. For example, the bounded array indexing property requires analysis of the possible values of array indices, which Mypy does not support; both the correct local port direction and valid hierarchical reference properties involve complex analysis of signals and components, which is challenging to encode in Mypy's type system.

Mypy does not Account for Path Conditions during Analysis – Hardware generators can also include if-statements to conditionally generate different hardware instances based on the given generator parameters. We refer to the if-conditions required to model a structural connection or behavior as *path conditions.* Path conditions can significantly affect the analysis of hardware generators. For example, the for loop body on line 28-38 in Figure 3 (b) connects different signals based on the if-condition on line 29 and line 36. If-conditions can affect the result of the array index analysis because the possible values of an array index may depend on the if-condition. Mypy does not account for path conditions by design and therefore cannot perform the analysis described above.

4 CHECKING GENERATOR PROPERTIES WITH SYMBOLIC ELABORATION

In this section, we propose a novel technique, *symbolic elaboration*, to address the limitations of off-the-shelf static type checkers. We observe that important generator properties can be encoded into integer constraints and that such constraints can be solved by a satisfiability modulo theory (SMT) solver. We design and implement a *symbolic elaborator* to perform symbolic elaboration, which statically analyzes the given hardware generator, builds an abstract generator model, constructs the integer constraints corresponding to the properties, and solves them using the Z3 SMT solver [21]. Compared to Mypy, the symbolic elaborator is able to reason about path conditions and statically verify all four properties in §2.2.

4.1 Building Abstract Generator Models

In this section we discuss how to build the abstract generator model based on a given target generator. We will use Figure 4 (a) and (b) as an example and demonstrate how to build an abstract adder model based on an example adder generator.

```
1  T = TypeVar("T", bound=Bits)
2
3  class Adder(Component, Generic[T]):
4    def __init__(s, Width:Type[T]) -> None:
5      ...
6    def construct(s, Width:Type[T]) -> None:
7      n       = get_nbits(Width)
8      s.a     = InPort(Width)
9      s.b     = InPort(Width)
10     s.out   = OutPort(mk_bits(n+1))
11     s.fa    = [FullAdder() \
12                for _ in range(n)]
13     s.carry = Wire(mk_bits(n+1))
14     s.sum   = Wire(Width)
15
16     for i in range(n):
17       if i >= 0:
18         connect(s.carry[i+1],s.fa[i].cout)
19         ...
20       if i == 0:
21         connect(s.carry[i],0)
22
23     @update
24     def upblk() -> None:
25       s.out @= concat(s.carry[n], s.sum)
```

(a) Target adder generator

Adder Symbol Table

Name	Type	DefCond
Width	Bits; generator arg	true
n	int; to_value(Width)	true
i	**int; $0 \le$ i $<$ n**	**i ≥ 0**

Adder Abstract Generator Model

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

(b) Symbolic elaboration results

Property: Bounded Array Index

`s.carry[i+1]`

Array Length:	n+1
Index Expression:	i+1
Use Condition:	i $\ge$ 0

$$\neg\,(0 \le i+1 < n+1)\ \wedge\ (i \ge 0)$$
$$\wedge\ (0 \le i < n)\ \wedge\ (n = Width)$$

Property: Matching Bitwidth

`out@=cat(carry[n],sum)`

LHS Width:	n+1
RHS Width:	1+to_value(Width)
Use Condition:	true

$$\neg\,(n+1 = 1 + Width)$$
$$\wedge\ (n = Width)$$

(c) Properties to Constraints

Figure 4: Symbolic Elaboration of an Adder. (a) target adder generator to be symbolically elaborated (identical to Adder in Figure 3 (b)); (b) symbolic elaboration results: the abstract generator model of adder and the symbol table; dark red line indicates the state of the symbol table when the elaborator is processing line 19; (c) translation of bounded array index and bitwidth matching properties into integer constraints.

Reasoning about Generator Argument Arithmetics – The key feature that distinguishes symbolic elaboration from static type checkers like Mypy is the ability to precisely reason about generator argument arithmetics. Figure 4 (b) shows the symbol table (which keeps track of active symbols the elaborator has encountered) and the abstract generator model (which records all active attributes of the current generator). The abstract generator model demonstrates that generator attributes can be potentially generic over a symbol or any arithmetics on symbols and concrete numbers. For example, the `s.out` port has a bitwidth derived from the arithmetics between the argument `Width` and an integer one. This arithmetic operation is preserved and will be translated into SMT-solvable constraints when checking properties of the abstract model.

Accounting for Path Conditions – In contrary to static type checkers like Mypy, path conditions are first-class citizens in symbolic elaboration: every entry in the symbol table or the abstract generator model has a *definition condition* (or defcond) which is the path condition up to the entry's point of definition. The symbolic elaborator maintains path conditions by pushing the if-condition or its negation to the *PathConds* stack before visiting the statements in the if- or else-branch. It pops the condition from the *PathConds* stack after the if-statement visit has finished. When the elaborator registers an entry with the symbol table or the abstract generator model, it constructs a boolean expression of the conjunction of all conditions in *PathConds* and use it as the definition condition of the entry. For example, in Figure 4 (b), the dark red entry for loop induction variable i has a definition condition $i \ge 0$ because it is under the if-branch of the if-condition on line 17.

4.2 Checking Properties of Abstract Generator Models

In this section we discuss how to encode hardware generator properties and path conditions into integer constraints that the Z3 SMT solver can verify. We will use Figure 4 (c) as an example and demonstrate how to check the bounded array index and matching bitwidth properties on an example adder generator.

Translating If-Conditions and Numeric Expressions – We use the term numeric expressions to refer to the arithmetics between symbols (i.e., generator arguments and loop induction variables) and concrete integers. Translating numeric and boolean expressions into Z3 expressions is straightforward because (1) generator arguments have a one to one correspondence to integer variables in Z3; (2) for-loop induction variables correspond to integer variables with two constraints on its lower and upper bound (e.g., induction variable `i` in `for i in range(n)` has constraints `i >= 0` and `i < n`); (3) conversion between bitwidth types and integers using `get_nbits` and `mk_bits` functions can be handled by adding one constraint that asserts the two symbols involved in the conversion are equal; (4) the Python binding of Z3 also offers integer constants and common binary arithmetic/comparison operations over integers.

Encoding the clog2 Function – One difficulty in encoding numeric expressions is that the `clog2` operation (`clog2(n)` = $\lceil log_2 n \rceil$) does not have a straightforward encoding in SMT solvers like Z3. We address this issue in two ways. We first add constant folding support so that `clog2` operations on constant values can be evaluated and replaced with its result. For non-constant `clog2` operands, we encode the `clog2` operation as an *uninterpreted function* from an integer to an integer in Z3. Z3 makes no assumptions about an uninterpreted function F except that $x = y \Rightarrow F(x) = F(y)$. This encoding scheme makes sure that any verified generator properties related to `clog2` must be true (i.e., no false-negatives).

Verifying Generator Properties – After translating numeric and boolean expressions to Z3, it is straightforward to verify the generator properties. To verify the bounded array index property, we construct an integer constraint that asserts the index has values out of the array bound. More specifically, for index expression i

```
module Generator
{
  -- Statements
  stmt = Construct(arg* args, stmt* body)
       | If(bool_expr cond, stmt* body, stmt* orelse)
       | TmpVarAssign(string target, num_expr value)
       | AttrAssign(string target, inst value)
       | SignalAssign(sig_expr target, sig_expr value)
       | Connect(sig_expr u, sig_expr v)
       | For(string var, int start, int end, stmt* body)

  -- Numeric Expressions
  num_expr = GeneratorArgNum(string name)
           | InductionVarNum(string name,
                             int start, int end)
           | UnsizedNum(int num)
           | UniOpNum(num_uni_op op, num_expr value)
           | BinOpNum(num_expr left,
                      num_bin_op op,
                      num_expr right)
           | NumFromDataType(data_type t)

  num_uni_op = NumClog2

  num_bin_op = NumAdd | NumSub | NumMult

  -- Types
  data_type = GeneratorArgDataType(string name)
            | DataTypeFromNum(num_expr bitwidth)

  type = ComponentType(string comp)
       | DataType(data_type t)

  -- Signal Expressions
  sig_expr = CurrentGenerator()
           | AttributeAccess(sig_expr value, string attr)
           | ArrayIndex(sig_expr value, num_expr index)
           | VectorIndex(sig_expr value, num_expr index)
           | BinOpSig(sig_expr left, sig_bin_op op,
                      sig_expr right)
           | Concat(sig_expr* args)

  sig_bin_op = Add | Sub | Mult | LShift

  -- Boolean Expressions
  bool_expr = NumCompare(num_expr left,
                         bool_cmp_op op,
                         num_expr right)
            | BinOpBool(bool_expr left,
                        bool_bin_op op,
                        bool_expr right)
            | UniOpBool(bool_uni_op op, bool_expr value)

  bool_cmp_op = BoolCmpEq | BoolCmpLt
  bool_bin_op = BoolAnd | BoolOr | BoolXor
  bool_uni_op = BoolNot

  -- Component/Signal Instantiation
  inst = CompInst(string comp, arg* args, int* dims)
       | SignalInst(data_type bitwidth, int* dims)

  -- Construction Arguments
  arg = Argument(string name, type t)
}
```

Figure 5: Core Generator Modeling Syntax Targeted by the Symbolic Elaborator.

and array length *len* with definition condition *def* and under use condition *use*, integer constraint Equation 2 will be checked by Z3 for counter examples (as shown in Figure 4 (c)). To verify if bitwidth u is the same as v under definition condition *def* and use condition *use*, we solve (1) (as shown in Figure 4 (c)). To verify the correct port direction or the valid hierarchical access property, we solve the conjunction of definition condition *def* and use condition *use* Equation 3 and consult the symbolic elaboration results to check if the port direction or hierarchical name is valid. It is worth noting that if Z3 finds $def \wedge use$ unsatisfiable, checking the target property is unnecessary because the definitions of symbols and signals in the target property are not available at the point of use.

$$(u \neq v) \wedge def \wedge use \tag{1}$$

$$\neg(0 \leq i \wedge i < len) \wedge def \wedge use \tag{2}$$

$$def \wedge use \tag{3}$$

If Z3 finds Equation 1 or Equation 2 unsatisfiable, we have obtained a proof that $u = v$ or i does not cause out-of-bound accesses for all possible values of integer variables. Otherwise we have obtained a counter example which corresponds to a set of symbol values that lead to design issues to be fixed.

4.3 Symbolic Elaboration Implementation

We implement a symbolic elaborator prototype that targets a subset of PyMTL3 hardware modeling DSL. Our implementation requires type annotations for generator parameters as shown in Figure 3 (b). The supported modeling syntax allows efficient static analysis and is also expressive enough to model most hardware generators. Figure 5 shows the core components of the supported modeling syntax in the Zephyr abstract syntax description language [33]. The core syntax is designed to be similar to the Python 3 syntax [25] and serve as a straightforward target for PyMTL3 hardware generators.

Statements – The root of the generator abstract syntax tree is a `Construct` node, which corresponds to the `construct` method of each hardware generator. We explain the following three syntax nodes because they interact with the elaborator data structure in important ways: (1) `If` nodes corresponds to an if-else statement in the generator (e.g., line 17,20 of Figure 4 (a)); this node has an if-condition of boolean expressions which are used to derive path conditions; (2) `For` nodes corresponds to a for-loop in the generator (e.g., line 16 of Figure 4 (a)); this node introduces a new name constrained by the star and end of the for-loop under the current path condition; (3) `AttrAssign` nodes indicate the addition of signal and sub-component attributes to the current generator (e.g., line 8-14 of Figure 4 (a)); this node adds an attribute to the current generator under the current path condition.

Signal Expressions – Signal expressions reference signals in the current generator and are common operands of arithmetic operations and structural connections (e.g., `s.fa[i].cout` and `concat(s.carry[n], s.sum)` in Figure 3 (b) are both signal expressions). Common signal expressions include attribute accesses (`AttributeAccess`), indexing into a signal array (`ArrayIndex`), indexing into a signal (`VectorIndex`), binary arithmetic operations between signals (`BinOpSig`), and signal concatenation (`Concat`).

Numeric Expressions – Numeric Expressions represent integers used in hardware generators. Common numeric expressions include generator arguments that have an integer type (`GeneratorArgNum`), for-loop induction variables (`InductionVarNum`), and integer literals (`UnsizedNum`). The numeric expressions also keep track of the binary (`BinOpNum`) or unary (`UniOpNum`) operations between numeric expressions. This enables precise reasoning about numeric values, which is critical in matching bitwidth analysis.

Boolean Expressions – Boolean expressions are used in if-conditions. The syntax in Figure 5 assumes that only comparisons between numeric expressions can be used in the if-conditions.

Algorithm 1 Core Symbolic Elaboration Algorithm. defcond=definition condition; G_i=set of input ports of generator G; G_o=set of output ports of generator G; G_s=set of subgenerators of generator G (i.e., generators instantiated inside G); G_p=$G_i \vee G_o$ (set of all ports of generator G).

```
Require: T: AST node of the generator under elaboration.
Require: P: Path conditions (set of bool_expr).
Require: N: Symbols encountered (set of num_expr).
Require: G: Generator (set of pairs (attribute name, inst)).
Ensure: Return type of AST node T; none if T is a statement.
1:  function SymbElab($T, P, N, G$)    ▷ Args passed by reference.
2:      if T is Construct then
3:          for all $stmt \in T.body$ do
4:              SymbElab($T, P, N \vee T.args, G$)
5:      if T is If then
6:          for all $t \in T.body$ do
7:              SymbElab($t, P \vee T.cond, N, G$)
8:          for all $t \in T.orelse$ do
9:              SymbElab($t, P \vee \neg T.cond, N, G$)
10:     if T is For then
11:         for all $t \in T.body$ do
12:             SymbElab($t, P, N \vee (T.var, T.start, T.end), G$)
13:     if T is AttrAssign then    ▷ $v|P$: v only valid if P holds
14:         $v \leftarrow$ SymbElab($T.value, P, N, G$)
15:         $G \leftarrow G \vee (T.target, v)|P$
16:     if T is GeneratorArgNum then
17:         $N \leftarrow N \vee T.name|P$
18:         return GeneratorArgNum(T.name)
19:     if T is ArrayIndex then
20:         $I \leftarrow$ bitwidth of SymbElab($T.index, P, N, G$)
21:         $L \leftarrow$ length of SymbElab($T.value, P, N, G$)
22:         $C \leftarrow$ defcond of $I$ and $L$ via $N \vee G$
23:         if $\neg(0 <= I < L)$ solvable w.r.t $C \wedge P$ then
24:             Report out-of-bound array index at $T$
25:     if T is SignalAssign,Connect,BinOpSig then
26:         ▷ Assume two operands are $T.left$ and $T.right$.    ◁
27:         $L \leftarrow$ bitwidth of SymbElab($T.left, P, N, G$)
28:         $R \leftarrow$ bitwidth of SymbElab($T.right, P, N, G$)
29:         $C \leftarrow$ defcond of $L$ and $R$ via $N \vee G$
30:         if $\neg(L = R)$ solvable w.r.t $C \wedge P$ then
31:             Report bitwidth mismatch at $T$
32:         if T is SignalAssign then
33:             $t \leftarrow$ SymbElab($T.left, P, N, G$)
34:             if $\exists H \in G_s : t \in G_i \vee H_o$ w.r.t $C \wedge P$ then
35:                 Report incorrect port direction at $T$
36:     if T is AttributeAccess then
37:         $V \leftarrow$ SymbElab($T.value, P, N, G$)
38:         $C \leftarrow$ defcond of $V$ via $N \vee G$
39:         if $V \in G_s \wedge T.attr \notin V_p$ w.r.t $C \wedge P$ then
40:             Report invalid hierarchical reference at $T$
```

However, it is straightforward to extend the syntax to support if-conditions with signal values (often used in behavioral modeling). Boolean expressions enable the precise reasoning of array index values, which is critical to detecting out-of-bound array indices.

Core Symbolic Elaboration Algorithm – Algorithm 1 shows the core algorithm of symbolic elaboration which is based on the traversal of the target generator's abstract syntax tree (AST; syntax defined in Figure 5). The symbolic elaborator holds three book-keeping data structures during elaboration; as the elaborator walks through the generator AST, it keeps tracks of all symbols derived from the generator's arguments and for-loop induction variables (N), maintains the current path condition based on the if-conditions encountered and the if-else branches (P), and records all attributes added to the generator (G). While verifying the properties discussed in §2.2, the algorithm generates a *path constraint* which is the conjunction between the current use condition and the definition conditions of names and expressions under verification ($C \wedge P$). The algorithm checks for out-of-bound array indices by searching for index values less than zero or larger than or equal to the length under the path constraint with the SMT solver; similarly it checks for bitwidth mismatches by searching for mismatched left and right bitwidths under the path constraint; checks for correct port directions and valid hierarchical names only invoke the SMT solver to verify the path constraint is satisfiable and verify the properties by looking up attributes in the bookkeeping data structures.

Table 2: Evaluated Hardware Generators

	Generator LoC	Instance LoC	Instance
LFSR	31	23	32-bit register
Gray Enc./Dec.	42	76	32-bit input
Priority Enc.	45	79	32-bit input
RR. Arbiter	49	88	8 requesters
FIFO Queue	125	174	32-bit 2-element queue
Divider	172	535	32-bit datapath
Processor	903	2135	single-core RV32-IM
CGRA	1170	4004	8×8 32-bit PE array

LoC (lines of code) reported by `cloc`; lines exclude comments and blanks. The upper/lower segment includes standard hardware IPs/standalone designs. Enc./Dec.: encoder/decoder. RR: round-robin.

5 EVALUATION

In this section, we describe how we evaluate the prototype implementation of symbolic elaboration and the evaluation results. In order to quantitatively evaluate symbolic elaboration's static checking capabilities, we implement a mutation-based abstract syntax tree (AST) fuzzer that randomly injects six common categories of bugs that we learned from actual hardware generator development. We randomly inject mutations into eight design generators using the AST fuzzer and compare when the bug is detected (earlier is better): ahead of time (early), elaboration time (middle), or simulation time (late). We compare our SE prototype to PyMTL3 with and without the Mypy static type checker.

5.1 Evaluation Designs

Our evaluation of symbolic elaboration uses eight hardware generators shown in Table 2. LFSR, Gray encoder/decoder, priority encoder, round-robin arbiter, and FIFO queue are commonly used hardware IPs. Divider, processor, and CGRA represent three standalone hardware designs: an integer divider, a RISC-V processor, and an elastic coarse-grain reconfigurable array [12]. The divider generator generates a radix-4 iterative divider for a given data path width. The processor generator generates one or more RV32-IM five-stage cores. The CGRA generator generates a reconfigurable processing element array with elastic flow control for the given dimension. The hardware generators are used in the bug detection evaluation and their corresponding instances are used in the simulation performance evaluation.

We choose these eight designs because they form a representative suite of hardware generators that includes both commonly used IPs and standalone hardware designs. On the one hand, standard hardware IPs are generally harder to check statically because such generators are heavily parameterized and sometimes customized for specific parameter combinations. On the other hand, hardware designers are more interested in SE's checking capabilities on real designs because it is much closer to the designer's use cases.

5.2 Mutation-Based Abstract Syntax Tree Fuzzer

To evaluate the effectiveness of symbolic elaboration's static checking capabilities, we implement a mutation-based AST fuzzer to inject mutations to designs in §5.1. The AST fuzzer is capable of injecting six kinds of mutations that we categorized from 249 git commits in the development and testing of the three standalone design generators. The AST fuzzer mutates part of the target syntactic construct to produce a syntactically correct but potentially semantically flawed hardware generator. The AST fuzzer recognizes the following syntactic constructs: constant integers, explicitly sized constants, signal slicing, accessing attributes of a component, referencing an object by identifier, and all arithmetic and boolean operations. *Each mutation stresses symbolic elaboration's ability to verify one of the generator properties discussed in §2.2.* It is worth noting that the AST fuzzer may inject mutations that are benign (not a bug), which we classify as a bug not detected (§5.3).

Bitwidth Mutation – Based on our experiences reviewing hardware development git commits, bitwidth mismatching is a common category of design bugs in generators. To perform a bitwidth mutation, The AST fuzzer searches for signal bitwidth declaration statements, slicing operations, and constants and mutates the bitwidth to force a bitwidth mismatch. This category of bugs stresses symbolic elaboration's ability to verify the *bitwidth matching* property.

Component Attribute Mutation – Dynamic HDLs rely heavily on accessing component attributes to construct hardware programmatically. It is often easy for designers to mix the name of one attribute with the other, and component attribute mutation aims to inject this kind of bug. More specifically, The AST fuzzer looks for an attribute name in attribute access (get or set) and replaces it with the name of another attribute from the same object. This mutation stresses symbolic elaboration's ability to verify the *valid hierarchical reference* property.

Port Direction Mutation – Incorrect port direction is another common category of design issues. The AST fuzzer introduces port direction bugs by flipping the direction of one port in the given design. This mutation stresses symbolic elaboration's ability to verify the *local port direction* property.

Name Expression Mutation – Name expression mutation focuses on changing the variable identifiers to model typos that are common in the git commits we reviewed. However, mutating a variable name will almost certainly generate references to nonexistent variables, which is trivial. To avoid referencing nonexistent variables (which is a trivial bug), The AST fuzzer keeps track of all available variable names in the current scope and only replaces the target name with a name in the current scope. This mutation stresses the general robustness of the prototype SE implementation.

Attribute Base Mutation – Attribute base mutation is a category of bugs where hardware designers remove the base object from attribute access expressions (e.g., `s.out` becomes `out`). Most of these bugs lead to accessing nonexistent variables. However, when this bug appears on the left-hand side of an assignment expression, the mutated syntax will create a temporary variable which can only be detected at simulation time. This mutation stresses the general robustness of the prototype SE implementation.

Functionality Mutation – Finally, the AST fuzzer implements functionality mutation by flipping constant values and arithmetic operators in generators. Unlike other mutations, some functional bugs cannot be detected through type checking (e.g., `a-b` type checks if `a+b` already type checks). Therefore, a test suite of high code coverage is required to achieve a high detection rate in this category. But other cases of functionality mutation which changes the constants in an array index expression can be detected because symbolic elaboration verifies the *bounded array indexing* property.

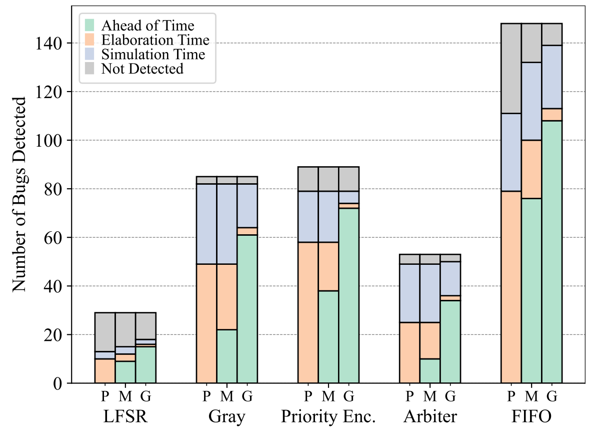
5.3 Bug Detection

In this section, we evaluate the bug detection effectiveness of PyMTL3, Mypy+PyMTL3, and SE+PyMTL3 on eight design generators. We randomly inject mutations into a generator using the AST fuzzer (one mutation at a time) and then evaluate if PyMTL3, Mypy+PyMTL3, and SE+PyMTL3 are able to detect the injected mutation. This evaluation compares the effectiveness of symbolic elaboration against state-of-the-art elaboration-time checks (PyMTL3) and an off-the-shelf static type checker (Mypy).

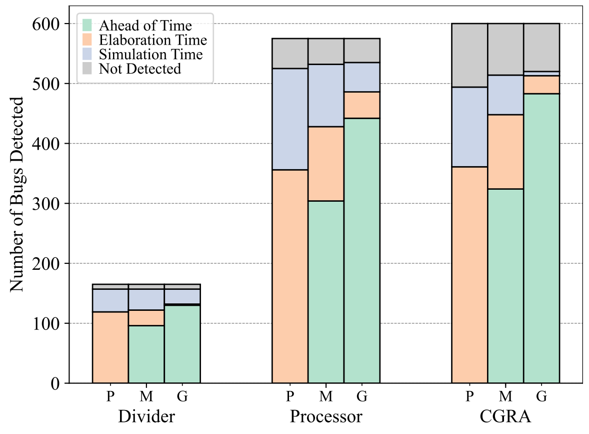
Figure 6 shows the number of bugs detected by PyMTL3, Mypy+PyMTL3, and SE+PyMTL3 at each stage: ahead of time (statically), during elaboration, during simulation, or not detected (including benign mutations). A syntax mutation may not be detected because the mutation does not change the generator's functionality or the simulation test vectors do not reach 100% coverage. We can see that *symbolic elaboration detects the same number of more bugs than PyMTL3 and Mypy+PyMTL3.* On average, symbolic elaboration detects 90.6% of the injected syntax mutations, which is slightly higher than PyMTL3 (86.6%) and Mypy+PyMTL3 (89.4%). This demonstrates the effectiveness of symbolic elaboration, which is able to detect bugs missed by simulation (simulation test vector may not achieve 100% coverage). Figure 6 also shows that *symbolic elaboration is able to detect significantly more bugs ahead of time than Mypy.* On average, symbolic elaboration detects 77.1% of syntax mutations ahead of time, whereas Mypy only detects 50.4%. This confirms that the symbolic elaborator is a better approach to statically check generators than existing static type checkers.

6 RELATED WORK

Symbolic elaboration is partly inspired by existing symbolic execution techniques. Symbolic execution generally applies constraint



(a) Bug Detection Results on Common IPs



(b) Bug Detection Results on Standalone Designs

Figure 6: Bug Detection Results – (a) and (b) show the number of bugs detected by PyMTL3 (P), Mypy+PyMTL3 (M), and GT-HDL (G) at different stages. The more bugs detected ahead of time the better.

solving techniques to reason about path conditions and is commonly used in program analysis activities including software testing, software vulnerability detection, and security analysis [8, 28, 31]. Traditional concrete execution of programs requires concrete test inputs, and each execution is limited to one control flow. Symbolic execution is able to explore multiple conditional paths in parallel and keep track of the constraints applied to the input symbols along the path as a series of boolean expressions. Symbolic execution often leverages a satisfiability modulo theories (SMT) solver to verify whether certain program properties have been violated and whether some paths are feasible [6]. If the SMT solver proves that some properties are violated, or a path is feasible, it can generate a feasible solution to the boolean expressions, which can be mapped back to a concrete test input that triggers the offending violations or exercises certain branches.

The symbolic elaboration technique is similar to symbolic execution techniques in two ways: (1) symbolic elaboration is a purely static analysis technique and does not support arbitrary code execution; (2) symbolic elaboration also heavily relies on constraint solving and SMT solvers. However, symbolic elaboration is different from symbolic execution in that (1) symbolic elaboration leverages constraint solving to reason about path conditions, matching bitwidths, and bounded array indexing, whereas symbolic execution typically uses constraint solving to trigger rarely explored program control paths that might include program bugs; (2) symbolic elaboration maintains much fewer symbolic states (only generator arguments in numeric expressions are symbolic) than symbolic execution, which generally maintains symbolic states for memory locations to generate accurate analysis results [16].

Salama et al. propose to use constraint solving to detect consistency issues in Verilog generator interconnects [27]. They define Featherlight Verilog, a hardware modeling language inspired by Verilog syntax, as the target of their consistency analysis. Both Featherlight Verilog and symbolic elaboration encodes signal bitwidths and array indices as integer constraints and use an SMT solver to detect inconsistencies among bitwidths and indices. However, symbolic elaboration targets more hardware generator properties and more aspects of hardware modeling. Featherlight Verilog specifically targets bitwidth and array index inconsistencies in structural modeling, whereas symbolic elaboration targets bitwidths, array indexing, port directions, and hierarchical references in both structural and behavioral modeling.

Rondon et al. propose liquid types, a type system that infers dependent types to aide static detections of out-of-bound array indexing errors [26]. Similar to symbolic elaboration, liquid types leverage a constraint solver to verify the safety of array accesses. Both SE and liquid type try to reduce programmer type annotation efforts by inferring precise types whenever possible. However, symbolic elaboration focuses on inferring types for dynamic HDLs and can statically verify generator properties discussed in §2.2. In contrast, liquid types focuses on protecting array indexing accesses in traditional functional programming languages.

7 CONCLUSION

In this paper, we propose a novel static checking technique called symbolic elaboration to shorten the design-debug cycles in dynamic HDLs. Symbolic elaboration targets the hardware generator modeling syntax and translates generator properties into integer constraints solvable by SMT solvers. It is able to verify generator properties including matching bitwidths, correct local port directions, bounded array indexing, and valid hierarchical references. Our evaluation of a prototype symbolic elaborator demonstrates a 53.0% improvement in its ability to statically identify design bugs comparing to an off-the-shelf static type checker.

ACKNOWLEDGMENTS

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, a research gift from Xilinx, Inc., and donations from Intel.

REFERENCES

[1] 2009. IEEE Standard VHDL Language Reference Manual. Online Webpage. https://ieeexplore.ieee.org/document/5981354.

[2] 2017. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. Online Webpage. https://ieeexplore.ieee.org/document/8299595.

[3] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[4] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. Cλash: Structural Descriptions of Synchronous Hardware Using Haskell. *Euromicro Conf. on Digital System Design (DSD)* (Sep 2010).

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. *DAC* (Jun 2012).

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* (2018).

[7] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. *ICFP* (Sep 1998).

[8] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. *International Conference on Software Engineering (ICSE)* (2011).

[9] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. *FPL* (Sep 2017).

[10] Jan Decaluwe. 2004. MyHDL: A Python-based HDL. *Linux Journal* (Nov 2004).

[11] Shai Fine and Avi Ziv. 2003. Coverage Directed Test Generation for Functional Verification using Bayesian Networks. *DAC* (Jun 2003).

[12] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic CGRAs. *FPGA* (Feb 2013).

[13] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. *ICCAD* (Nov 2017).

[14] Shunning Jiang, Yanghui Ou, Peitian Pan, Kaishuo Cheng, Yixiao Zhang, and Christopher Batten. 2021. PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies. *IEEE Design Test* 38 (Apr 2021), 53–61. Issue 2.

[15] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. 2020. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro* 40 (May 2020), 58–66. Issue 4.

[16] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* (1976).

[17] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. *ICCAD* (Nov 2018).

[18] Jukka Lehtosalo. 2017 (accessed Nov., 2021). Mypy - Optional Static Typing for Python. Online Webpage. (2017 (accessed Nov., 2021)). http://mypy-lang.org.

[19] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *MICRO* (Dec 2014).

[20] 2013 (accessed Nov., 2021). Migen: A Python Toolbox For Building Complex Digital Hardware. Online Webpage. (2013 (accessed Nov., 2021)). https://m-labs.hk/gateware/migen/.

[21] Leonardo De Moura and Niklaj Bjørner. 2008. Z3: an Efficient SMT Solver. *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Mar 2008).

[22] Amir Nahir, Avi Ziv, and Subrat Panda. 2012. Optimizing Test-Generation to the Execution Platform. *Asia and South Pacific Design Automation Conference (ASP-DAC)* (Jan 2012).

[23] Matthew Naylor and Simon Moore. 2015. A Generic Synthesisable Test Bench. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)* (Sep 2015).

[24] Rishiyur Nikhil. 2004. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)* (Jun 2004).

[25] Python. 2021 (accessed Apr., 2022). Python Documentation - Abstract Syntax Trees. Online Webpage. (2021 (accessed Apr., 2022)). https://docs.python.org/3.7/library/ast.html.

[26] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. *PLDI* (2008).

[27] Cherif Salama, Gregory Malecha, Walid Taha, Jim Grundy, and John O'Leary. 2011. Static Consistency Checking for Verilog Wire Interconnects. *Higher-Order and Symbolic Computation* (2011).

[28] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). *IEEE Symposium on Security and privacy* (2010).

[29] Ofer Shacham, Megan Wachs, Andrew Danowitz, Sameh Galal, John Brunhaver, Wajahat Qadeer, Sabarish Sankaranarayanan, Artem Vassilev, Stephen Richardson, and Mark Horowitz. 2012. Avoiding Game Over: Bringing Design to the Next Level. *DAC* (Jun 2012).

[30] 2013 (accessed Nov., 2021). SpinalHDL. Online Webpage. (2013 (accessed Nov., 2021)). https://spinalhdl.github.io/SpinalDoc-RTD/.

[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing through Selective Symbolic Execution. *Network and Distributed System Security Symposium* (2016).

[32] Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. *Summit on Advances in Programming Languages (SNAPL)* (May 2019).

[33] Daniel C. Wang, Andrew W. appel, and Jeff L. Korn. 1997. The Zephyr Abstract Syntax Description Language. *Conf. on Domain-Specific Languages (DSL)* (Oct 1997).