Towards Gradually Typed Hardware Description Languages

Peitian Pan, Shunning Jiang, Yanghui Ou, Christopher Batten
Cornell University
Ithaca, NY, USA

ABSTRACT

Recent research in hardware development methodologies has argued for sophisticated hardware generators and dynamically typed high-level components to improve the hardware design and verification productivity. In this talk, we describe our on-going work towards gradually typed HDLs to realize such productivity benefits through powerful static type checking and safe and performant composition of mixed-typed components.

1 MOTIVATION

Specialized hardware tend to have high non-recurring engineering (NRE) costs that hinder the development of promising hardware systems. Recent research addresses the high NRE costs in two ways: (1) parametrized hardware generators to maximize design reuse [1, 3, 10, 12, 16, 18, 20] and (2) dynamically typed high-level components to enable polymorphic test harnesses, reference models, and cycle-approximate models [7, 11–15]. Unfortunately, several challenges have prevented state-of-the-art HDLs from realizing these productivity benefits. **First, existing HDLs suffer from a long design-debug cycle because they generally fail to statically type check hardware generators.** For example, instead of verifying matching bitwidths statically, most existing HDLs delay bitwidth checks among connections until hardware instances have been generated and bitwidth parameters have been resolved into concrete values. **Second, disciplined composition of mixed-typed components is difficult.** Statically typed components are guaranteed to drive well-typed values on output ports, as long as all input port values have the expected types. In contrast, dynamically typed components can drive potentially ill-typed values on output ports because the input values can be ill-typed, or there might be a type error in the component. **Third, modern dynamically typed HDLs sacrifice simulation performance to ensure a disciplined composition of mixed-typed components.** Modern HDLs that support dynamically typed components generally insert simulation-time checks to ensure safe interoperation between components [12]. However, this approach incurs performance overhead and fails to leverage the static type information.

2 A SPECTRUM OF EXISTING HDLS

Table 1 shows that existing HDLs either (1) do not support sophisticated hardware generators and dynamically typed components or (2) have long design-debug cycles due to late type checks.

Traditional HDLs – Verilog/SystemVerilog [9] and VHDL [8] are traditional statically typed HDLs. Traditional HDLs enforce type invariants by type checking the elaborated hierarchy of hardware instances, which happens in the middle of a hardware design cycle.

High-Level Statically Typed HDLs – Bluespec SystemVerilog [17] allows static type checking of a generator to discover potential design issues early in the design cycle. However, Bluespec is not able to detect bitwidth mismatches in vector slicing operations and defers this check to elaboration. Cλash [2] is a Haskell dialect for hardware development. It benefits from Haskell's static type system and can type check the generators before elaboration.

Embedded Statically Typed HDLs – Chisel [3] and SpinalHDL [19] are embedded in Scala, a statically typed programming language. Lava [4] is embedded in Haskell and supports sophisticated generators. Embedded statically typed HDLs mainly enforce type invariants by checking the elaborated hardware instances.

Embedded Dynamically Typed HDLs – PyRTL [5], MyHDL [6], PyMTL [14], and PyMTL3 [12] are embedded in Python, a dynamically typed programming language. Almost all existing embedded dynamically typed HDLs lack static type checking capabilities, and most of them do not perform elaboration- nor simulation-time type checks. Instead, type errors typically occur when an ill-typed object is *used*, which causes difficulties for debugging.

Gradually Typed HDLs – GT-HDLs achieve the best of both statically and dynamically typed HDLs. Instead of only allowing either statically or dynamically typed components, GT-HDLs support the safe interoperation of mixed-typed components. Similar to high-level statically typed HDLs, GT-HDLs improve design productivity by performing static type checks on hardware generators to shorten the design-debug cycle. Similar to embedded dynamically typed HDLs, GT-HDLs are flexible and allow dynamically typed high-level components to improve verification productivity. GT-HDLs can also improve verification productivity by improving simulation performance using the available static type information.

3 RESEARCH DIRECTIONS FOR GT-HDLS

We discuss three techniques that address the challenges in §1. We envision that a GT-HDL implementation will incorporate these techniques to boost hardware design and verification productivity.

Static Type Checking for Generators – GT-HDLs need powerful static type checking capabilities to ensure a short design-debug cycle. Existing HDLs either do not perform static checks at all (e.g., embedded dynamically typed HDLs) or require intimate knowledge of advanced type systems (e.g., high-level/embedded statically typed HDLs) that are foreign to most hardware designers.

We propose to build static type checkers that specifically target the HDL syntax of synthesizable hardware generators and translate critical hardware generator properties into integer constraints solvable by a satisfiability modulo theory (SMT) solver. We assume

LATTE '23, March 26, 2023, Vancouver, BC, Canada

HDLs	**Sophisticated Generators**	**Dyn. Typed High-Level Components**	**How are Type Invariants Enforced?** Type check generators	Type check instances	Type check in simulations	**When Type Checks Occur in Design Cycle**
Traditional HDLs Verilog/SystemVerilog, VHDL			○	●	○	Middle
High-Level Stat. Typed HDLs Bluespec SystemVerilog, Cλash	✓		●	◒	○	Early/Middle
Embedded Stat. Typed HDLs Chisel, SpinalHDL, Lava	✓		◒	●	○	Early/Middle
Embedded Dyn. Typed HDLs PyRTL, MyHDL, PyMTL, PyMTL3	✓	✓	○	●	●	Middle/Late
Embedded Grad. Typed HDLs GT-HDL*	✓	✓	●	◒	◒	Early/Middle/Late

Sophisticated generators: generators that programmatically generate hardware instances. Earlier type checks in the design cycle leads to lower bug-fixing costs. ●/◒/○ : almost all/some/no invariants enforced; Stats./Dyn./Grad. typed: statically/dynamically/gradually typed; *: our proposal.

Table 1: A Spectrum of Existing Hardware Description Languages

that hardware generators take non-negative integer parameters, which correspond to integer variables in the generated constraints.

For example, a repeater generator generates circuit that duplicates the n-bit input X times to form the output, where n and X are parameters. The proposed static type checker first propagates the symbolic bitwidths to all signals in the generator. It then builds the following bitwidth constraint for the syntax construct that concatenates the X copies of n-bit input to the output:

$$\underbrace{n + n + \ldots + n}_{X \text{ times}} = n \times X \qquad (1)$$

If the SMT solver finds ¬(1) unsatisfiable, the checker has established the proof that both sides of the concatenation has matching bitwidth; if the solver finds a satisfiable assignment to variables n and X, the checker has found a counterexample that triggers a potential design issue in the generator. The above trivial example can be generalized to verify more properties including bounded array indices and matching bitwidths on slicing operations.

Safe Mixed-Typed Component Composition – In response to the second challenge in §1, we propose elaboration-time and simulation-time type checks. Elaboration-time type checks verify the hardware generator parameters against its type signature to prevent admitting ill-typed parameters. Simulation-time type checks are inserted to every signal assignment that crosses the mixed-typed component boundary and verify if LHS and RHS of the assignment have matching bitwidth. Figure 1 shows a composition of a statically typed divider and a dynamically typed test bench. The light blue signals are the targets of simulation-time type checks. Only three simulation-time checks are necessary for correctness because the statically typed domain is guaranteed to generate well-typed operands as long as its input is well-typed.

Type-Based Simulation Optimizations – GT-HDLs enable mixed-typed component compositions and can potentially benefit from the static type information of hardware components. To demonstrate how a GT-HDL implementation can leverage static type information for simulation performance, we describe *signal coalescing*, a technique that reduces the number of signal assignments in simulation. To simulate the behavior of a circuit, the simulator of an HDL generally presents a group of connected signals using the *net* data structure [12], where one *writer* signal continuously

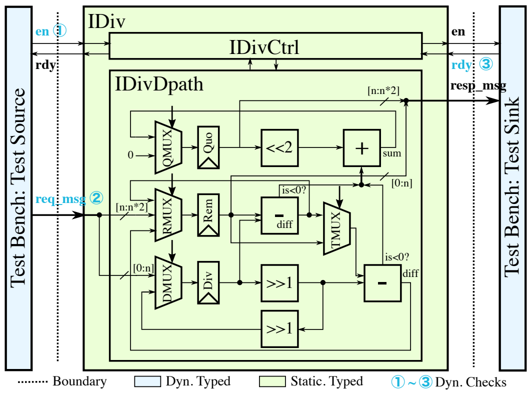

Figure 1: A Mixed-Typed Composition in GT-HDL

drives values to zero or more *readers*. In an unoptimized implementation, the continuous update is generally implemented as per-cycle signal assignments from the writer to all readers, which include a simulation-time type check to prevent a dynamically typed writer from injecting ill-typed values. Instead of performing actual signal assignments, signal coalescing sets up a reference from the group of dynamically typed readers to the dynamically typed writer to eliminate unnecessary assignments.

4 CONCLUSION

Hardware designers have benefited from the type safety of statically typed HDLs and the flexibility of dynamically typed HDLs. However, existing HDLs remain either statically or dynamically typed, which limits the peak designer productivity. In this paper, we propose GT-HDLs to achieve the best of both worlds. We believe GT-HDLs are a compelling solution to realize the productivity benefits of advances in hardware design methodology research. We hope this paper will spark interests from the methodology research community to fully realize the potential of GT-HDLs.

ACKNOWLEDGMENT

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] 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.

[2] 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).

[3] 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).

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

[5] 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).

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

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

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

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

[10] 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).

[11] 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.

[12] 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.

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

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

[15] 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).

[16] 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).

[17] 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).

[18] 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).

[19] 2013 (accessed Nov., 2021). SpinalHDL. Online Webpage. (2013 (accessed Nov., 2021)). https://spinalhdl.github.io/SpinalDoc-RTD/.

[20] 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).