A Guide for Rapid Creation of New HDLs

Sakshi Garg*, Sheng-Hong Wang†, Hunter James Coffman‡, Jose Renau§
Dept. of Computer Science and Engineering
University of California, Santa Cruz
Email: {*sgarg3, †swang203, ‡hcoffman, §renau}@ucsc.edu

***Abstract*—We present a guide for quickly building and testing a new HDL. Upcoming HDLs have a compiler designed specifically for each of them, which means duplicated efforts. The proposed guidelines minimize these efforts. New language developers can leverage existing compiler infrastructures to interface with the new HDL. This interfacing would enable developers to use LEC without any reference compiler. The proposed method also leads to converting the newly developed HDL to any other language supported by the interfaced compiler. Also, this system would ensure higher trust in the language and the interfaces.**

I. Introduction

The new HDLs have till date been developed with their corresponding compiler infrastructure [1]–[3]. Recently, there has been a push for new hardware design flows like LLHD [4], CIRCT [5] and LiveHD [6] that can handle multiple languages. Interfacing any new HDL with such multi-language flows can save tremendous effort of the new language developer.

Using a multi-language compiler to develop any new HDL (let us say *X-lang*) is not a straightforward task. This is because no formal verification tool supports the grammar of the new language. As in [7], the developers could use reference compilers to verify the flow. For HDLs, this flow verification is done using Logical equivalence Check (LEC) [8]. Since the language is new, no reference compiler is available. So there is no way to verify the correctness of the output. Moreover, higher coverage means higher confidence in the system [9], [10]; but a new language does not have a diverse program set to ensure good coverage. Additionally, a random code generator for *X-lang* would not be useful because of the absence of a reference compiler.

To address these issues, this paper explains a methodology to develop a new language (*X-lang*) using one of the multi-language supporting flows (*Y-flow*). Without using any reference compiler or building an entire compiler flow, the *X-lang* developer could leverage a *Y-flow* to construct an *X-lang* compiler flow. This paper explains the passes needed to develop the tests for testing this system of *X-lang*and *Y-flow*. The methodology to create these tests, such that *X-lang* generates correct results is discussed in the paper. These tests would identify the bugs in *X-lang* to *Y-flow* interfaced system. Also, as mentioned before, the widely practised method to develop trust in a system is by having a high coverage. This paper addresses the problem of low coverage as well. More importantly, the methodology of this paper would provide reliability and trust earlier in the design flow and thus a shorter development cycle.

We propose the following tests and flows:

1) ***X-lang to X-lang LEC***: This LEC test (shown in figure 2) consists of a method to generate back *X-lang* from *Y-flow* in addition to translating the HDL into *Y-flow*.
2) ***X-lang* to Verilog tests**: Some subsets of the language are surprisingly difficult to get correct, but they can be checked for logical equivalence by generating equivalent expressions in the new HDL and Verilog as shown in Figure 1.
3) **Random/Synthetic program generation**

These proposed tests will enable the *X-lang* developer to detect any bugs in the entire flow obtained after leveraging *Y-flow* for *X-lang* generation. No reference compiler would be required for the verification of the flow. Also, the propose tests and flows would help in increasing coverage and thus building trust in the system. Additionally, these flows would enable conversion of other *Y-flow* supported languages to *X-lang*, using the *X-lang* generation. Our methodoloy will save the *X-lang* developers' time and effort. Language developers can then deliver a more equipped language faster to the open-source community for use. The details of

these proposed flows along with an example implementation will be discussed in detail in section III.

Having *X-lang* generation has some other advantages not explored in this paper. It allows you to have a large code base in your language. This allows to "absorb" other projects and increase the code base.

II. Related Work

The problem of not having a unified workflow for all languages is present in Software [11] as well as Hardware community.

In Hardware community, there has been a push for new hardware design flows like LLHD [4], CIRCT and LiveHD [6] that can handle multiple languages. These flows use generic IR constructs and aim to avoid redundancy at different stages of a design flow. With such standardized compilers, a new language developer would not need to create an entire compiler flow for a language. Current limitation is that these flows provide the output netlist in Verilog and/or their own representation like *Netlist LLHD* [4]. since each compiler till now has been language specific, so if you start designing a new HDL, you initially do not have a reference compiler. Thus you cannot check the validity of your new language using standards like LEC.

III. Methodology

Every language is processed into different IRs by its respective compiler [1], [12]. This paper proposes a methodology to compose a new HDL (*X-lang*) and consolidate it with a pre-existing compiler (*Y-flow*). Let the *X-lang* be Pyrope [13] and *Y-flow* be LiveHD for implementation in this paper. The focus of the paper is on the approach to devise tests for robust *X-lang* development using *Y-flow*. This system would also check that the new HDL interfacing generates expected results by using checks like LEC. We aim to maximize the coverage of *X-lang* constructs to increase reliability. As an impact, the correctness of the compiler flow is also checked in the process.

We coded the following to implement the proposed system.

A. X-lang input interfacing

The newly developed HDL needs to be interfaced with the Y-flow. To ensure multi-language support, Y-flows have language independent IRs. Thus we need to translate X-lang to a syntax parse tree and further to language independent IR.

Once the input code has been processed to the language independent IR [4], [14], the design can go through all the available compiler passes in *Y-flow* [15]. Thus, the new HDL designer does not need to spend effort and time for the compiler system.

In our implementation of this pass, we developed *inou.pyrope*[1]. This pass translates Pyrope to a syntax parse tree and further to the Language neutral AST in LiveHD [6] called LNAST [14].

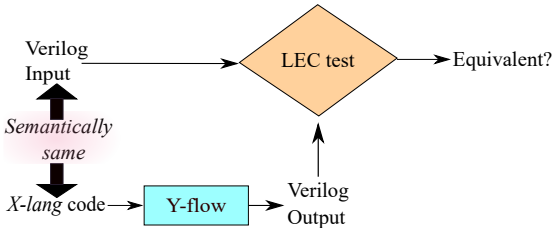

Fig. 1. A LEC check by generating the *X-lang* code semantically equivalent to a Verilog code.

B. X-lang Generation

This pass generates *X-lang* back from the *Y-flow* IR. *X-lang* generation has two main advantages. First, It allows swift conversion of other *Y-flow* supported HDLs to *X-lang*. Second, it enables leveraging the existing Verilog code base testing and using the most reliable Hardware language check i.e LEC.

We propose two LEC based flows using above passes. First in Figure 1, both input and output Verilogs should pass LEC. Second is *X-lang to X-lang LEC*, as shown in Figure 2. In this flow, the two generated Verilog files should pass LEC too. This would validate the conversions relating to the new HDL. The validity of the translations "HDL to AST" and "AST back to HDL" could be verified using boundary cancellation [16].

LiveHD provides a low-level LNAST (LL-LNAST) at the end of all compiler passes. So, we implemented a pass[2] to translate LL-LNAST to

[1]https://github.com/masc-ucsc/livehd/tree/master/inou/pyrope

[2]https://github.com/masc-ucsc/livehd/blob/master/inou/code_gen/code_gen.cpp

Pyrope. All the constructs from LN were taken care of to be embedded in Pyrope.

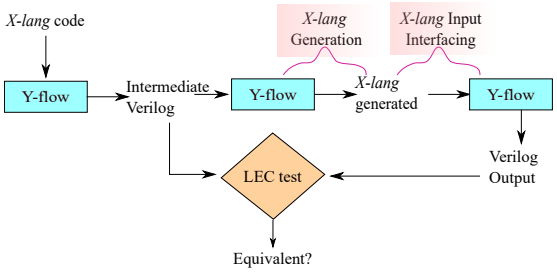

Fig. 2. *X-lang to X-lang LEC* flow.

Both, *X-lang Input Interfacing* and *X-lang generation*, are required to carry out the *X-lang to X-lang LEC* flow.

C. Random/Synthetic program generation

To further ensure complete coverage for reliable language production, Random/synthetic program generation is required along with above proposed passes. *X-lang* developer could generate programs from existing Verilog designs and the new system could be checked against them. As an example, BOOM [17] could be generated in the new HDL and with the help of proposed system, a LEC could be performed to test the newly developed language. This would cover a major section of the features of the HDL. Further, remaining coverage could be checked using targeted programs to include remaining constructs and potential failures in the language are expected to be highlighted in the process. This process can be used to develop and test any new HDL and reliably launch it within short development cycle.

D. X-lang to Verilog tests

This part generates semantically same designs in Verilog and *X-lang* as seen in Figure 1. *X-lang* to Verilog tests are needed because *X-lang* generation is not enough to maximize coverage reliably. To understand, suppose there is a pathological bug where *X-lang* ”(x * -2)” gets incorrectly translated to ”(x * 2)” in *Y-flow* IR. The *Y-flow* internal representation may never generate this test maybe because constants are always first. Thus ”(-2 * x)” is used as the canonical form, not ”(x * -2)”. This means that *X-lang to X-lang LEC* would not capture this bug. The proposed solution to this problem is to generate random programs in *X-lang* and at the same time generate the equivalent Verilog code(Figure 1). This is possible to do for large subsets of the language like typical control flow statements. For some more complex cases, we could also generate a basic regression, as discussed.

***X-lang* to Verilog tests** proposed have some similarities with CPU verification. We have a set of random program generation, and we have a way to test them. These tests should be guided to cover different aspects of the language features. In addition, there is a set of human generated tests to check the flow. This would ensure higher trust in the language and the interfaces.

Y-flow (LiveHD in our case) provides translation of the design to Verilog from IR (LGraph [18] in LiveHD). This Verilog can be used for LEC tests to validate the new proposed HDL.

There are some constraints on the proposed methodology. To start, the new language should support the complete set of *Y-flow* internal representation. For example, *Y-flow* may support negative edge flops while the original designers did not want to support them. This would enable interfacing of multiple HDLs with each other through *Y-flow*. Nevertheless, some features like tri-states may not be supported. In such cases, Verilog tests that trigger these features in the *Y-flow* IR can not be used.

IV. Evaluation

Developing a new language as well as interfacing it with pre-existing compiler is an error prone process. In this section we evaluate how by using the proposed flows, you can detect bugs in the entire compiler flow. Also, by sharing our results of the Pyrope-LiveHD flow formation, we evaluate how you can increase coverage. Note that since it is a work in progress, this section shares the preliminary evaluation.

A. Bug detection

The proposed flow helped us detect bugs in Pyrope as well as LiveHD. Below are examples of the different types of bugs detected during different testing phases.

1) Bugs detected by Pyrope input interfacing pass (as in section III-A):

a) During our LEC test from figure 1, we detected a missing shift operation for the LiveHD operator get_mask. Our tests successfully pointed out to this.

2) Bugs detected by Pyrope generation pass (as in section III-B): We implemented some assertions based on our understanding of LNAST grammar in the pass[3]. This helped us catch many issues with the LiveHD pass in which LGraph is converted to LNAST.

a) LGraph could provide a decimal value as well as a Hex value for a mask. Added this support in Pyrope generation pass.
b) After encountering some bugs, extra care was taken to handle the LNAST iterators.
c) This pass helped us understand syntactic rules of LNAST each time we encountered a bug. As an example, a function call's 1st child in LNAST could be a temporary variable. Temp vars are used in LNAST to store references to some entities like tuples.
d) Handling some shift operators.
e) Handling muxes.
f) Unary vs n-ary operators.
g) Correct Indentation in output file for better readability.
h) When Shift right arithmetic accesses a variable from get_mask, issue in destination pin of get_mask node was detected.

3) Bugs detected by manual regression suite: We wrote over a hundred types of programs trying to cover the constructs to catch any issues in our flow. These programs included all control flow statements, tuples, and other LiveHD specific operators like get_mask. Different configurations like multiple varied nesting in if-else statements were kept in mind while designing programs.

a) Detected some deadlock issues while running the flow in Figure 2. All the detected bugs were reported to developers and duly rectified by them.
b) Reported a bug in LiveHD regarding incorrect destination pin of get_mask operator.
c) Different operators for handling tuples according to their presence on rhs or lhs of an assignment.
d) Bug in Depth preorder traversal operation of LNAST notified.

We reported some deadlocks in LiveHD while experimenting with converting other languages like chisel/FIRRTL to pyrope. By default, this paper didn't expect LiveHD to have bugs and any issue caught would be in the input/output then. In case of bug detected in LiveHD, narrowed and targeted tests could be performed to debug the intenal steps. Some constructs that required most of the debugging were Function calls, tuple handling and handling operators involving masks.

B. Coverage of the new HDL

A high coverage on the language features have always been a measure of reliabilty and testing [9], [10]. We hypothesize that coverage of the new HDL Pyrope can be maximized and checked using the proposed flow. As mentioned in III, random/synthetic program generation can be used to generate test cases to cover all the constructs of Pyrope. For this paper, we manually designed test cases. We propose that a synthetic program generator could considerably increase coverage of the language and provide a reliable flow. Further work would include the evaluation of HDL coverage using LNAST. We would like to target LNAST for this analysis due to its language neutral approach. This would enable coverage analysis for any language added to *Y-flow* in future [19].

V. Conclusion

New HDL developers can leverage existing multi-language compiler flows for the HDL by following the proposed guidelines. This would provide them with a reliable system in a short span of time and lowered efforts as compared to designing a dedicated compiler. The proposed method would detect issues in the new HDL as well as the pre-existing flow, if any. Also, it enables conversion to other languages supported by the compiler.

Acknowledgments

This work has been supported by the Center for Research in Open Source Software (CROSS) at UC Santa Cruz. This material is based upon work

[3]https://github.com/masc-ucsc/livehd/blob/master/inou/code_gen/code_gen.cpp

supported by, or in part by, the Army Research Laboratory and the Army Research Office under contract/grant W911NF1910466. Also, thanks to all the reviewers for their continuous feedback and support. Thanks to Mark Zakharov for his contribution to semantically same program generation.

REFERENCES

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[2] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017, pp. 1–7.

[3] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 209–216.

[4] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: A multi-level intermediate representation for hardware description languages," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 258–271.

[5] llvm, "llvm/circt: Circuit ir compilers and tools." [Online]. Available: https://github.com/llvm/circt

[6] S.-H. Wang, R. T. Possignolo, H. B. Skinner, and J. Renau, "LiveHD: A Productive Live Hardware Development Flow," *IEEE Micro*, vol. 40, no. 4, pp. 67–75, 2020.

[7] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[8] S. Gayathri and T. Taranath, "Rtl synthesis of case study using design compiler," in *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*. IEEE, 2017, pp. 1–7.

[9] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[10] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[11] C. Wimmer and T. Würthinger, "Truffle: a self-optimizing runtime system," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, 2012, pp. 13–14.

[12] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 209–216.

[13] Sheng-Hong Wang, Haven Skinner, Sakshi Garg, Hunter Coffman, Kenneth Mayer, Akash Sridhar, Rafael T. Possignolo, and Jose Renau, "Pyrope," http://masc.soe.ucsc.edu/livehd/pyrope/, Online; accessed on 16 August 2021.

[14] S.-H. Wang, A. Sridhar, and J. Renau, "LNAST: a language neutral intermediate representation for hardware description languages," in *Open-Source EDA Technology, Proceedings of the Second Workshop on*, ser. WOSET'19, Nov. 2019.

[15] Wikipedia, "Types of compiler optimizations." [Online]. Available: https://en.wikipedia.org/wiki/Optimizing_compiler#Types_of_optimization

[16] J. T. Perconti and A. Ahmed, "Verifying an open compiler using multi-language semantics," in *European Symposium on Programming Languages and Systems*. Springer, 2014, pp. 128–148.

[17] K. Asanović, D. Patterson, and C. Celio, "The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," University of California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-167, Jun. 2015.

[18] S.-H. Wang, R. T. Possignolo, Q. Chen, R. Ganpati, and J. Renau, "LGraph: a unified data model and api for productive open-source hardware design," in *Open-Source EDA Technology, Proceedings of the Second Workshop on*, ser. WOSET'19, Nov. 2019.

[19] T. Pool, A. R. Gregersen, and V. Vojdani, "Trufflereloader: A low-overhead language-neutral reloader," in *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2016, pp. 1–10.