LGraph: A Unified Data Model and API for Productive Open-Source Hardware Design

Sheng-Hong Wang, Rafael Trapani Possignolo, Qian Chen, Rohan Ganpati, Jose Renau
Dept. of Computer Science and Engineering,
University of California Santa Cruz.
http://masc.cse.ucsc.edu

ABSTRACT

We present LGraph, a unified data model and API for productive open-source hardware design. It is inspired by live incremental synthesis and aims to provide a fast infrastructure for a productive hardware development flow. In this paper, LGraph is described both from the perspective of a hardware designer as well as an EDA software developer. Key features of LGraph include a unified data model and API, a fast memory mapped library design, integration with third-party tools and hierarchical design traversal for third-party tools such as OpenTimer, Mockturtle and ABC.

1. INTRODUCTION

There is a resurgence in hardware design due to the degradation of performance, power, and area with hardware specialization in multiple areas. A new momentum of design innovation would come from open-source EDA tools and highly productive hardware design flows. Hardware designers want a fast design flow to iterate between synthesis and its analysis. EDA tool developers, in particular from the open-source community, want to work with a common model and API to focus on the tool's algorithm development.

Ideally, a productive hardware design flow should have a very short design iteration period. This helps the designers to fast implement the new design idea based on the feedback from an interactive environment. Whereas, in a traditional design flow, it is common for designers to wait for hours or even days to obtain the design result.

Open-source EDA tools also plays a vital role for hardware innovation as fellow researchers and hardware developers could contribute their novelty without facing licensing constraints. In recent years, research works such as DATC [14], qflow [1], VTR [22] and OpenROAD [4] focus on integrating tools of different design stages into a single RTL-to-GDSII flow. Some of the single-stage tools are ABC [7], Mockturtle [24], and Yosys [26] for logic synthesis, OpenTimer [11], OpenSTA [13] for static timing analysis, RePlAce [9] and NTUPlace3 [8] for placement and NCTU-GR [18] and TritonRoute [15] for routing.

These works on integrating open-source tools have shown the potential of building a tapeout-ready flow. Despite the correctness of these design flows, they are still far from being ideal. One important source of issues is the lack of a common data model and APIs. Individual tools are developed using different data structures which largely raises the integration difficulty. Moreover, tools not developed using a common data model end up replicating code and efforts. For example, almost every tool implements its own netlist parser. This code replication further causes a non-negligible portion on the flow execution time. To make matters worse, not all tools implement standards equally, causing compatibility issues.

There are several sources for the slow hardware compilation flow including design elaboration, logic synthesis, timing analysis, placement, and routing. The state-of-art incremental technique like SMatch [21] has been applied to provide an interactive experience but is limited to synthesis. However, EDA tools are IO heavy applications and re-parsing the netlist and libraries to the internal data structure of these tools additionally takes a large portion on the flow's run time. As mentioned in [19], it would take Yosys [26] tens of seconds to parse a reasonably large RTL file, which leads to a less productive design experience. The situation is even worse when the project goes into debug or optimization phase. Although changes applied in multiple flow iterations are small, designers have to wait for the same re-parsing time repeatedly.

Performing design synthesis and static timing analysis in a hierarchical manner is a key feature for productivity in the hardware design flow. However, as mentioned in [10], many open-source logic synthesis and STA tools such as OpenTimer, Mockturtle, and ABC lack hierarchical design support as compared to industrial tools.

In order to optimize the whole design, the designer must first flatten the hierarchical design and then feed it to these tools. However, physically flattening every submodule during the logic synthesis phase would increase the complexity of the back-end physical synthesis. Furthermore, from the point of tool development, even if each tool implements the hierarchical feature, the insidious code replication among these tools still violates the DRY (don't repeat yourself) principle in software development.

In this paper, we present the enhanced version of LGraph, our attempt to build an infrastructure for productive hardware design flow. LGraph is inspired by the incremental synthesis technique LiveSynth [20], and stands for "live graph" for the goal of providing feedback from small design changes lively – within few seconds.

Following are the highlighted key features of the enhanced LGraph:

1. Unified data model/API: A unified data model and API in C++17 for digital circuits. LGraph is meant to represent netlists in different phases of the design flow from RTL to layout including simulation and code generation. The easy-to-use APIs largely reduce the design effort of tool developers. More importantly, the nature of the unified data model could prevent the possible code duplication and avoid parsing and generating the netlist between the internal stages of the RTL-to-GDSII flow.

2. Hierarchical design traversal: The hierarchical cross-module traversal ability of LGraph empowers the integrated third-party tools to run the core algorithm in a virtually flattened form. Therefore, LGraph could achieve the goal of global optimization implicitly without affecting the physical design phase.

3. Fast memory mapped library design: LGraph is built with a live interactive design flow in mind. We have designed a memory mapped C++17 library for fast netlist load/unload.

4. Third-party tools integration: LGraph is currently being integrated with some open-source tools such as Mockturtle, OpenTimer, and Yosys. Open-source EDA tool developers could leverage LGraph's succinct API and generic data structure to implement their algorithm and use the integrated third-party tools to complete the whole design flow.

2. LGRAPH

In this section, we first highlight the key features of LGraph for EDA tool developers: a fast memory mapped library design, selective developer-friendly APIs, the hierarchical attribute and traversal. We then discuss how LGraph could be integrated with other tools in the design flow and the integration of two third-party tools with LGraph. Lastly, we discuss some ideas to increase interoperability and speed of LGraph.

2.1 LGraph for EDA Developers

2.1.1 Fast Memory Mapped Library in C++17

Modern SoC design usually constitutes hundreds of millions, even billions of logic gates. To fast load/store such a large netlist, LGraph uses the memory mapping technique for fast persistence. The memory mapping technique will map a disk file directly to the virtual memory space and thus reduce the buffer copy operations. It has the speed advantage for large file processing [17]. We implement a fast memory mapped library called *mmap_lib* with basic data structures such as vector, hash map, bi-directional hash map, set, and tree. These fundamental containers form the skeleton of LGraph's storage code base. They were used extensively for constructing graph networks and attributes. As the program gets completed, LGraph's database is automatically synchronized to the disk by the OS.

2.1.2 Node, Pin, and Edge Construction

A single LGraph represents a single netlist module. LGraph is composed of nodes, node_pins, edges and tables of attributes. An LGraph node is affiliated with a node_type and each type defines different amounts of input and output node_pins. For example, a node can have 3 input pins and 2 output pins. Each of the IO pins can have many edges to other graph nodes. Every node_pin has an affiliated node_pid. A pair of driver_pin and sink_pin constitutes an edge. In the following API example, an edge is connected from a driver_pin (pid1) to a sink_pin (pid3). The bitwidth of the driver_pin determines the edge bitwidth.

```
auto node = lg->create_node(Node_Type_Op);
auto dpin = node.setup_driver_pin(1);
dpin.set_bits(8);
auto spin = node2.setup_sink_pin(3);
dpin.connect(spin);
```

2.1.3 LGraph Transversal

LGraph is a bidirectional graph representation supporting topological sort traversal in an input-forward and output-backward manner. If the order in which nodes are visited does not matter in the algorithm, developers could choose the fast iterator which would visit the next node in the cache line.

```
for (const auto &node:lg->forward()) {...}
for (const auto &node:lg->fast()) {...}
```

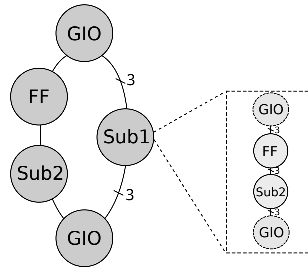

Figure 1: A hierarchical LGraphs. A sub-graph node is a sub-module instantiation. The hierarchical traversal will walk into the sub-graph structure.

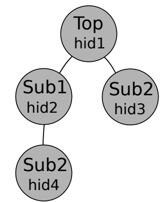

Figure 2: The tree hierarchical view for Figure 1. Each instantiation has a unique hid and hierarchical attribute table.

LGraph now supports hierarchical traversal. Each sub-module of a hierarchical design will be transformed into a new LGraph and represented as a sub-graph node in the parent module, as shown in Figure 1. If the hierarchical traversal is used, every time the iterator encounters a sub-graph node, it will load the sub-graph persistent tables to the memory and traverse the sub-graph recursively, ignoring the sub-graph input/outputs. This cross-module traversal treats the hierarchical netlist just like a flattened design. In this way, all integrated third-party tools could automatically achieve global design optimization or analysis by leveraging the LGraph hierarchical traversal feature.

```
for (const auto &node:lg->forward_hier()) {...}
```

2.2 LGraph Attribute Design

Design attribute stands for the characteristic given to a LGraph node or node_pin. For instance, the characteristic of a node name and node physical placement. Despite a single LGraph stands for a particular module, it could be instantiated multiple times, for example, the *sub2* node in Figure 1. In this case, same module could have different attribute at different hierarchy of the netlist. A good design of attribute structure should be able to represent both non-hierarchical and hierarchical characteristic.

2.2.1 Non-Hierarchical Attribute

Non-hierarchical LGraph attributes include pin_name, node_name and line of source code. Such properties should be the same across different LGraph instantiations. Two instantiations of the same LGraph module will have the exact same user-defined node name on every node. For example, in Figure 1, instantiations of a sub-graph 2 in both top and sub-graph 1 would maintain the same non-hierarchical attribute table.

```
node.set_name(std::string_view name);
```

2.2.2 Hierarchical Attribute

We introduced a new design of hierarchical LGraph attribute after an inspirational discussion with the author of FIRRTL. LGraph's hierarchical attribute is achieved by using a tree data structure to record the design hierarchy. In LGraph, every graph has a unique id (lg_id), every instantiation of a graph would form some nodes in the tree and every tree node is indexed by a unique hierarchical id (hid). As shown in Figure 2, we are able to identify a unique instantiation of a graph and generate its own hierarchical attribute table. An example of hierarchical attribute is wire-delay.

```
node_pin.set_delay(float delay);
```

2.3 3rd Party Tool Integration

The integration of third-party tools into LGraph is intuitive. Most tools have APIs to construct netlist in their internal data structure. Thus, we could first create an object of the tool in the LGraph program, traverse the LGraph netlist and use the tool's API to build an equivalent circuit on the fly inside the object. Then we make the object perform its main functions, for instance, synthesis. Finally, we map the tools data structure back into LGraph. Currently, Mockturtle and OpenTimer are being integrated into LGraph and there is an initial working prototype of the same.

2.3.1 Mockturtle

LGraph uses Mockturtle's library for LUT-based synthesis. We first partition combinational groups and map these groups from LGraph to Majority-Inverter Graph (MIG) [5] for synthesis. The synthesized MIG networks are then technology mapped to k-bit Lookup table (KLUT) networks and stitched back to LGraph.

2.3.2 OpenTimer

The synthesized LGraph will then use the integrated OpenTimer to perform timing analysis. Again we traverse the LGraph netlist and build the corresponding OpenTimer structure, compute timing inside the OpenTimer object and return the critical-path information.

2.4 Other Ongoing LGraph Work

2.4.1 Fast Verilog Spliter

We also implemented a very fast Verilog splitter. The splitter would parse through the design module and I/O definition to generate tokens, then split a large hierarchical design like BOOM into small Verilog files by module. The reason to split the large design is for a potential parallel elaboration optimization. The parsing process is very fast because we focus on the split and there is no construction of internal abstract syntax tree.

2.4.2 LNAST

Another project is the design of a language-neutral AST (LNAST) as the front end of LGraph. LNAST is meant to be the high-level intermediate representation (IR) to bridge different hardware description languages (HDLs) with LGraph. Currently LNAST supports Pyrope [23] and will be extended to Verilog and FIRRTL in the future.

2.5 LGraph for Design flow Users

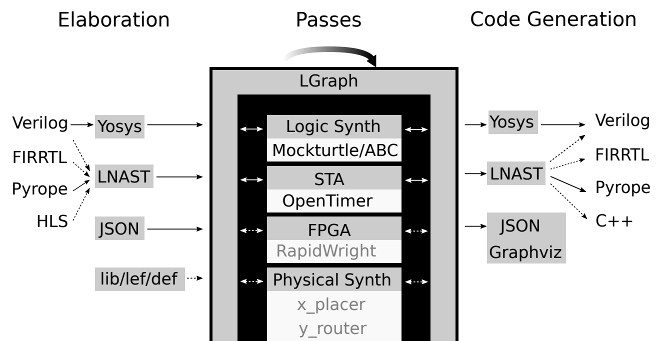

Figure 3: An overview of LiveHD flow. The dash-lines represent the future integration projects.

The ultimate goal of LGraph is building a live hardware development (LiveHD) flow for circuit designers. Figure 3 shows the overview of LiveHD. The LiveHD flow will start from the elaboration phase to generate an LGraph from HDLs. In the pass phase, LGraph interfaces with Mockturtle or ABC to perform technology mapping and logic synthesis, calls OpenTimer for STA, performs FPGA placement and routing using RapidWright [16]; and in the future, will apply physical synthesis by the integrated placer and router. Some other passes like dead code elimination and bitwidth optimization will further help achieve a more optimal LGraph. In the code generation phase, the conversion from LGraph to LNAST IR helps generate the optimized netlist and C++ code for simulation. There is also a plan to bridge LiveHD, FIRRTL [12] and Chisel [6].

3. EVALUATION

3.1 Setup

We compared LGraph's memory mapped library *mmap_-lib* with the C++17 standard library, Abseil C++ library [25], robin-map [2], and flat-hash-map [3]. We randomly generated 100k from a uniform random number generator to insert pairs of *uint32_t* key and value to hash map, then we erase the 100k elements randomly and measured the runtime. We perform both the vector and hash map flow 100 times and take the average execution time.

We also evaluate the scalability on the alpha LGraph-Mockturtle LUT synthesis flow and compare it with the Yosys-ABC synthesis flow [1] targeting Xilinx 7-series FPGAs. We used simple combinational chains of gates ranging from 1 to 50K serialized concatenations.

All experiments were run on a Intel Core i7-6700K CPU @ 4.20 GHz with 16 GB of memory, running Manjaro v5.2.8-1. Tools were compiled with gcc v9.1.0.

3.2 Results

3.2.1 LGraph Memory Mapping Library

We evaluated the access speed on various sizes of vectors and hash maps. LGraph's *mmap_lib::vector* is 39.1% faster than *std::vector* in our simple test, and LGraph's runtime scales better. Figure 4 shows the average run time for writing and reading the entire vector on both LGraph *mmap_lib::vector* and C++'s standard vector. The comparison result between C++ hash map implementations and *mmap_lib::map* is shown in Figure 5.

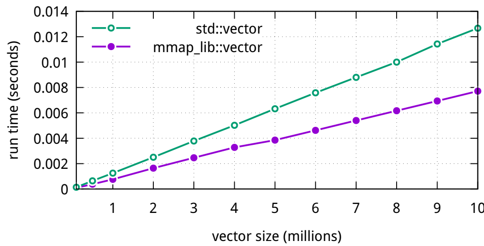

Figure 4: LGraph's mmap_lib::vector has faster runtime compared to the std::vector

The *mmap_lib::map* design shows a competitive speed among all competitors when hash table size is under 10 million and starts to outperform others when the table size is in the order of 10 million which is typically the size of a modern-day partitioned VLSI netlist. Not only does it have the fast container access time, LGraph's *mmap_lib* library also provides an extra advantage of data persistence which would be the key feature when developing live incremental VLSI flow as we dont have to re-parse the whole data again.

3.2.2 LGraph-Mockturtle LUT Synthesis Flow

We also looked into the LUT synthesis scalability for flows of LGraph-Mockturtle, Mockturtle only, and Yosys-ABC. The prototype of LGraph-Mockturtle LUT mapping flow starts from converting LGraphs to MIG networks, synthesizing and mapping them to KLUTs, and converting the KLUT networks back to LGraphs. The Mockturtle-only flow is almost the same but it excludes the conversion steps from and to LGraphs. For Yosys-ABC flow, we only measure the execution time from RTLIL to technology mapping, and to ABC LUT synthesis.

[1]Commands include (1) read_verilog (2) proc (3) techmap (4) abc -lut 4, only the time of (3) and (4) are measured

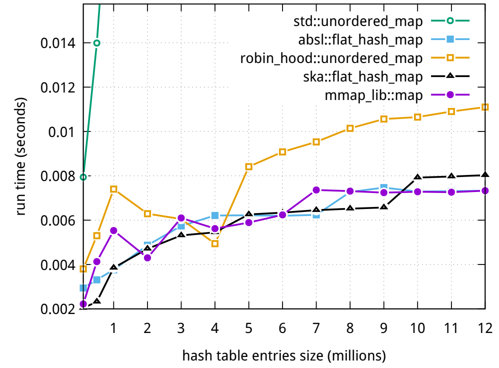

Figure 5: LGraph's mmap_lib::map is in par with best-in-class maps for entry sizes less than 10 million but faster for entry sizes in the order of 10 million

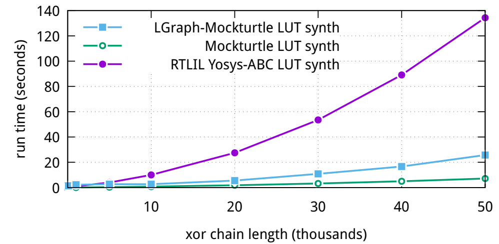

Figure 6: The LGraph-Mockturtle flow is faster than Yosys-ABC flow under all tested scenarios

Figure 6 compares the scalability of LUT synthesis among various flows such as LGraph-Mockturtle, Mockturtle only, and Yosys-ABC. Though the LGraph-Mockturtle flow is still in its early development stages, the run-time is certainly better than the Yosys-ABC flow. For a 50k combinational chain, it takes the Yosys-ABC flow 134 seconds to finish while only takes our LGraph-Mockturtle flow 25.7 seconds, an 80.8% speedup. When compared to the flow of Mockturtle only, there is an integration overhead in our flow; the main reason lies with the prototype implementation: there are some network copy operations in the integration between LGraph and Mockturtle, which take quite a bit of time. We are currently working on the algorithmic optimization to deal with this performance overhead.

4. CONCLUSIONS

In this paper, we present the enhanced LGraph as a fast infrastructure for open- source EDA developers and integrated design flow for hardware designers. New features include a fast memory mapped library to avoid

netlist re-parsing, the hierarchical traversal function to enable the integrated tools to handle hierarchical design support automatically, and a prototype flow of LGraph's integration with Mockturtle and OpenTimer.

Our results show that LGraph's memory-mapped vector is 39.1% faster than C++ standard library designs. The design of LGraph's memory-mapped hash map is comparable to the best C++ open-source implementations. We also show a working technology mapping flow with the integration of LGraph and Mockturtle for FPGA lut synthesis, which is 80.8% faster than Yosys-ABC flow and still have room to further speed up.

Future work includes integration with RapidWright [16] for FPGA placement and routing, interface for different HDLs and HLS from the new LNAST high-level IR, and optimizing LiveHD with technique of LiveSynth/SMatch [20, 21]for a productive open- source hardware design flow.

Acknowledgments

This work has been supported by the Center for Research in Open-Source Software (CROSS) at UC Santa Cruz, which is funded by a donation from Sage Weil and industry memberships. This work was also supported in part by the National Science Foundation under grant CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF

5. REFERENCES

[1] "Qflow," http://opencircuitdesign.com/qflow/, online; accessed on 20 August 2019.

[2] "robin-map," https://github.com/Tessil/robin-map, online; accessed on 28 August 2019.

[3] "ska-map," https://github.com/skarupke/flat_hash_map, online; accessed on 28 August 2019.

[4] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 76.

[5] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[6] 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.

[7] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.

[8] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "Ntuplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1228–1240, 2008.

[9] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "Replace: Advancing solution quality and routability validation in global placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[10] R. Friesenhahn and J. York, "Arl: Uts experiences in the free open-source vlsi eda landscape," Oct. 2018.

[11] T.-W. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool," in *Computer-Aided Design, Proceedings of the IEEE/ACM International Conference on*, ser. ICCAD'15. Piscataway, NJ, USA: IEEE Press, Nov. 2015, pp. 895–902.

[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] James Cherry, "OpenSTA," https://github.com/abk-openroad/OpenSTA, online; accessed on 5 September 2019.

[14] J. Jung, P.-Y. Lee, Y.-S. Wu, N. K. Darav, I. H.-R. Jiang, V. N. Kravets, L. Behjat, Y.-L. Li, and G.-J. Nam, "Datc rdf: Robust design flow database," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 872–873.

[15] A. B. Kahng, L. Wang, and B. Xu, "Tritonroute: an initial detailed router for advanced vlsi technologies," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[16] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 133–140.

[17] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang, "Mmap: Fast billion-scale graph computation on a pc via memory mapping," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 159–164.

[18] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "Nctu-gr 2.0: Multithreaded collision-aware global routing with bounded-length maze routing," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 32, no. 5, pp. 709–722, 2013.

[19] R. T. Possignolo, S. H. Wang, H. Skinner, and J. Renau, "LGraph: A multilanguage open-source database," in *Open-Source EDA Technology, Proceedings of the First Workshop on*, ser. WOSET'18, Oct. 2018.

[20] R. T. Possignolo and J. Renau, "LiveSynth: Towards an interactive synthesis flow," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 74.

[21] ——, "SMatch: Structural matching for fast resynthesis in fpgas," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 75.

[22] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial fpgas," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 1–4.

[23] H. Skinner, R. T. Possignolo, and J. Renau, "Liam: an actor based programming model for hdls." in *MEMOCODE*, 2017, pp. 185–188.

[24] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL logic synthesis libraries," May 2018, arXiv:1805.05121.

[25] T. Winters, "Non-atomic refactoring and software sustainability," in *2018 IEEE/ACM 2nd International Workshop on API Usage and Evolution (WAPI)*. IEEE, 2018, pp. 2–5.

[26] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/, 2019, online; accessed on 5 September 2019.