UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title
Boolector Interface with LGraph

Permalink
https://escholarship.org/uc/item/9rm3z9jw

Author
Kapp, Micaela Guerra

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

BOOLECTOR INTERFACE WITH LGRAPH

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Micaela Guerra Kapp

June 2022

The Thesis of Micaela Guerra Kapp
is approved:

Professor Jose Renau, Chair

Professor Scott Beamer

Professor Daniel Fremont

Peter Biehl
Vice Provost and Dean of Graduate Studies

Copyright © by

Micaela Guerra Kapp

2022

Table of Contents

List of Figures **v**

List of Tables **vi**

Abstract **viii**

Acknowledgments **ix**

1 INTRODUCTION **1**
- 1.1 Proposed Improvements 3
- 1.2 Related Work 4
  - 1.2.1 Logic Equivalence Checking 5
  - 1.2.2 Current Solvers 7

2 LIVEHD **9**
- 2.1 Importance of Live Development 9
- 2.2 Open-Source Third Party Tools 10
- 2.3 LiveHD Infrastructure 12

3 LGRAPH **14**
- 3.1 Overview 14
- 3.2 Nodes 15
- 3.3 Edges 17
- 3.4 Graph Traversals 17

4 SAT AND SMT SOLVERS **21**
- 4.1 SAT Solvers 21
- 4.2 SMT Solvers 24
  - 4.2.1 Eager Approach to SMT Solving 25

4.2.2 Lazy Approach to SMT Solving 26
4.3 Boolector 27

5 BOOLECTOR INTEGRATION **29**
5.1 LGraph 30
5.1.1 Node Types and Edges 30
5.1.2 Identifying LGraph Inputs and Outputs 33
5.2 Boolector 36
5.2.1 BTOR Format 38
5.2.2 C APIs 38
5.2.3 Example Usage 45
5.3 Integration of Boolector and LGraph 47
5.3.1 Creating Pass.LEC 47
5.3.2 Testing PASS.LEC 51

6 CONCLUSION AND FUTURE WORK **55**

Bibliography **57**

List of Figures

1.1 Proposed Changes 3
1.2 Generalized Synthesis Flow[30] 5
1.3 Compare Step of Equivalence Checking from Formality (Synopsys)[25] 6
1.4 Eager approach to SMT [31] 7

2.1 LiveHD Infrastructure[27] 12

3.1 LGraph of trivial.v using Graphviz[8] 16
3.2 Hierarchical traversal example[28] 19
3.3 Hierarchical Traversal Graph Structure[28] 20

4.1 Boolean CNF of Logic Gates 22
4.2 Miter Circuit for Logical Equivalence Checking 23
4.3 Eager approach to SMT[12] 25
4.4 Lazy approach to SMT [31] 27

5.1 Boolector Logic Flow Chart [4] 37
5.2 Example of Directed Acyclic Graph (DAG) 40

List of Tables

5.1 LGraph Node Attributes 31
5.2 LGraph Node Type Operations[27] 32
5.3 LGraph Node_pin Attributes 35
5.4 Boolector Node Types 42
5.5 LGraph and Boolector Node Type Equivalents 43
5.6 Boolector SAT Engine Typedefs[6] 44

Listings

3.1 Verilog Code of trivial.v 15
3.2 Forward traversal code example 17
3.3 Fast traversal code example 17
3.4 Backward traversal code example 18
5.1 Code to determine current node 30
5.2 Combining Node and Node_pin Attributes 33
5.3 Code to generate LGraph inputs 34
5.4 Code to generate LGraph outputs 34
5.5 Code to create Boolector Instance 38
5.6 Code to clone a Boolector Instance 39
5.7 Code to assert or release BoolectorNode 41
5.8 Code to delete Boolector Instance 41
5.9 Code to set Boolector SAT Solver 43
5.10 Code to simplify Boolector input 44
5.11 Code to call Boolector SAT 45
5.12 Code to simplify Boolector input 45
5.13 Define Boolector *btor and data type Example 46
5.14 BoolectorNode Usage Example 46
5.15 Boolector Formula Assertion Example 46
5.16 Release Boolector Usages 47
5.17 Setup Boolector for PASS.LEC 48
5.18 Boolector Formula Assertion Example 49
5.19 Code for returning solution to SAT call 50
5.20 Releasing Boolector usages from PASS.LEC 51
5.21 Commands to generate LGraphs trivial and trivial3 51
5.22 Command to test LEC pass 52
5.23 Output from PASS.LEC 52

Abstract

Boolector Interface with LGraph

by

Micaela Guerra Kapp

Modern electronics feature semiconductor chips which are incredibly sophisticated consisting of multi-millions of logic gates. A small change has the potential to produce disastrous consequences effecting project timelines and time to market. Developing these complex chips requires correct tools and verifiable designs during all stages of design and testing.

LiveHD is an open-source EDA tool for synthesis and simulation that provides quick feedback for small design changes. LGraph is the optimized, netlist, graph representation of LiveHD. To maintain correctness over optimizations and to aid in performing logic equivalence checking, the integration of an SMT solver, Boolector, with LGraph is proposed. This thesis outlines the process of developing the interface between Boolector and LGraph to produce a logic equivalence check (LEC) pass for LiveHD. Although full integration was not achieved, the current status of the pass, limitations and future work are discussed.

Acknowledgments

I would like to give my appreciation and thanks to my husband and family during the efforts to finish this work. Victor, thank you for your encouragement and taking on more; your love and emotional support did not go unnoticed. To my grandma and grandpa, thank you for your unconditional love, teachings, wisdom and for always watching over me.

To my graduate department advisor Emelye Neff, thank you for encouraging me to persevere and for believing in me and my abilities even when I couldn't. To Sheng-Hong Wang, thank you for the extra help at the beginning. Your patience and advice was very encouraging.

To my committee members Scott Beamer and Daniel Fremont, thank you for your flexibility, feedback and support. Your time and efforts are overwhelmingly appreciated.

To my thesis advisor and chair, Jose Renau, thank you for giving me the opportunity. Your kindness and willingness to work with me during my difficult times means more than you know. Thank you for always encouraging me to ask for help and the importance of not overthinking.

Chapter 1

INTRODUCTION

The expansion of the semiconductor industry has produced a need for more tools and technologies to continue fostering development. Semiconductor chips are incredibly complex but the tools for hardware development are lacking in modern features such as incremental synthesis and compilation. Small code changes result in hours of turnaround time. Improvements to Electronic Design Automation (EDA) tools from academia have solely focused on isolated steps of design while private organizations manage advancements through monetizing.

In industry, the software is usually considered intellectual property and the source code is kept confidential. Open source software fosters the bridge between academia, industry, and enthusiasts through community-oriented development. Contributors are able to equally use, modify, and distribute the software. Open

source EDA tools are crucial to the development of new hardware as licensing can be one of the many barriers to innovation.

Live Hardware Development (LiveHD) is an EDA tool that aims to produce synthesis and simulation results in a few seconds[27]. The tool consists of an open-source graph library using the incremental analysis model. LiveHD allows developers to maximize their productivity by providing quick feedback for small code changes, thus reducing the synthesis and simulation bottlenecks during the hardware design process. These stages of the design process often consist of slow and tedious operations with turnaround times from hours to days.

To ensure the Hardware Descriptive Language (HDL) representation remains the same during different stages of development, a Logical Equivalence Check (LEC) Pass is needed to improve LiveHD. This thesis describes the development of a LEC Pass and its current interface capabilities with LiveHD.

Chapter 2 will define the importance of live development, open-source tools, and their association with the LiveHD Infrastructure. Chapter 3 will explain the LGraph structure, it's usage and proposed improvements. Chapter 4 will explain the importance of SAT and SMT solvers with EDA tools. Chapter 5 will discuss the integration of Boolector with LiveHD and the results. Chapter 6 will summarize the thesis and outline future work.

1.1 Proposed Improvements

The integration of satisfiability (SAT) and satisfiability modulo theory (SMT) solvers within the LEC Pass will help streamline the flow by implementing equivalence checking within the tool. Figure 1.1 shows the current and proposed changes to the flow of LiveHD.

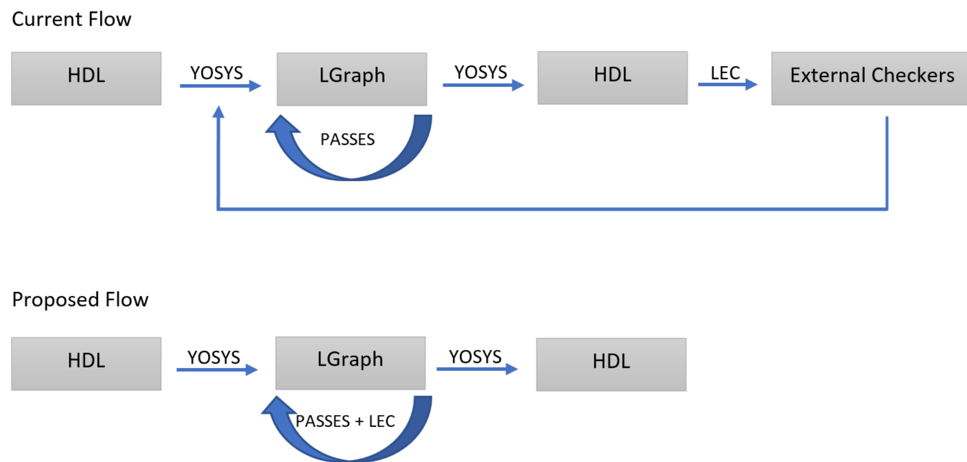

Figure 1.1: Proposed Changes

The current flow utilizes the external equivalence checker, Formality by Snyopsys, to perform equivalence checking. The process can be improved by eliminating the transformation from a LGraph to HDL representation before using external checkers for equivalence checking. Most importantly, the development of a LEC pass would check if a circuit is functionally the same before and after a transformation pass. Although the current process works, it is not efficient. The proposed

flow would streamline the work flow, allow synthesis transformations, verification, and equivalence checking to be integrated within the LiveHD framework, and eliminate the need for costly software.

The current status of the LEC Pass and Boolector interface is under development. Only partial integration was achieved as sequential circuits have not been fully integrated and all regression tests have not been passed. The status is discussed in more detail in Chapter 5. The source code for the current LEC Pass is found on the LiveHD Github [27] hand has been pushed to the master branch.

1.2 Related Work

Formal verification [24] is the process of checking if the behavior of a system described using a mathematical formal model satisfies a given property. Equivalence checking [11, 17] is a subset of formal verification. Equivalence checking, described by Mohnke et al. [17], is the problem of checking whether two circuit descriptions specify the same behavior and showing that modifications have not altered functional behavior. Modern logic equivalence checking (LEC) involves performing this process with multi-millions of gates in a single design [11, 17].

1.2.1 Logic Equivalence Checking

Goldberg et al. [9] argued that satisfiability is a more robust and flexible engine of Boolean reasoning for combinational equivalence checking than Binary Decision Diagrams (BDDs); their research showed a speedup of two orders of magnitude with the use of SAT. Incremental approaches include substitution, learning, and transformation based algorithms. Cheng and Huang [11] discuss that although Automatic Test Pattern Generation (ATPG) speeds up the verification process, they alone are still not the ideal method for exploring larger designs.

In a design flow, Logic Synthesis takes place the between the HDL and Netlist stages. A HDL design, such as Verilog or Pyrope, is synthesized to produce a RTL representation. A netlist is the RTL representation at a logic gate level. Figure 1.2 shows a generalized synthesis flow from HDL to netlist representations.

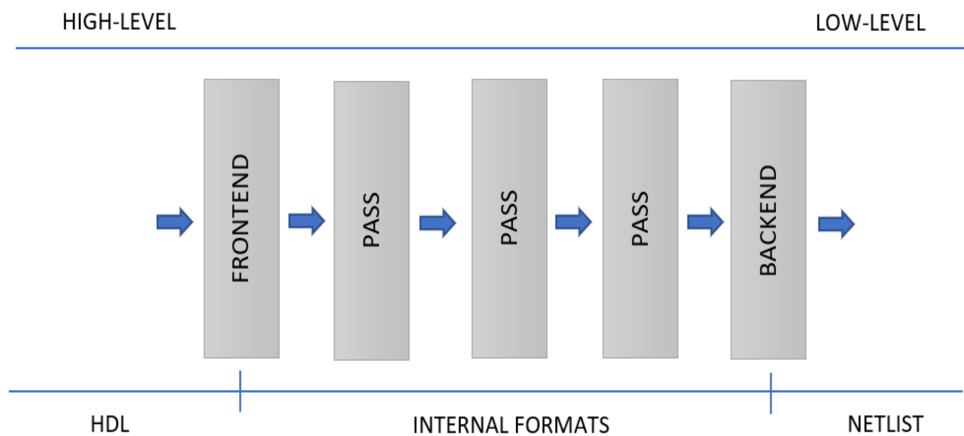

Figure 1.2: Generalized Synthesis Flow[30]

This transformation process is performed by an EDA tool requiring a logical equivalence check to ensure functionality remains the same between the two stages. Formality Equivalence Checking by Synopsys is a proprietary EDA tool that performs LEC and additional capabilities to aid developers. The comparison step of equivalence checking from Formality is shown in Figure 1.3.

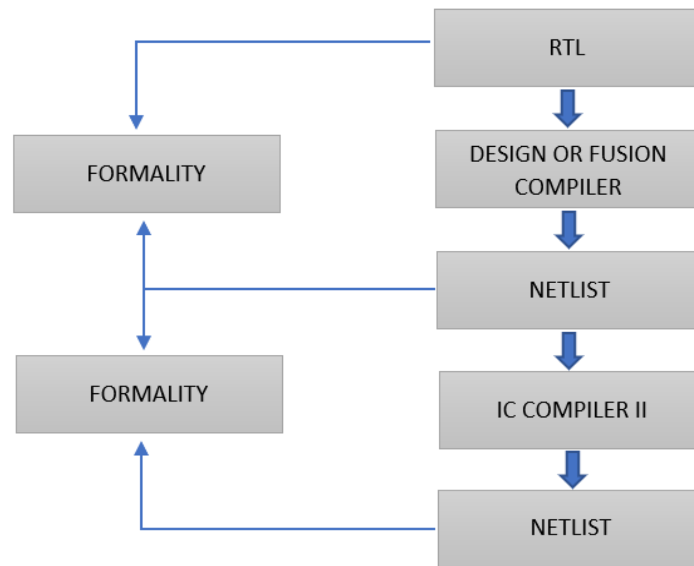

Figure 1.3: Compare Step of Equivalence Checking from Formality (Synopsys)[25]

Synopsys states that Formality uses formal, static techniques to determine if two versions of a design are functionally equivalent[25]. One drawback to proprietary software, is the uncertainty what specific SAT solver is used in Formality.

1.2.2 Current Solvers

SMT solvers offer an extension to the constraints that a SAT solver can solve with data types like bit vectors, arrays, strings, inequalities and more. There are two classes that categorize SMT solvers, Eager solvers and Lazy solvers. Eager solvers produce a larger formula that is passed to a SAT solver to determine satisfiability. Figure 1.4 illustrates an overview of the Eager approach to SMT solving.

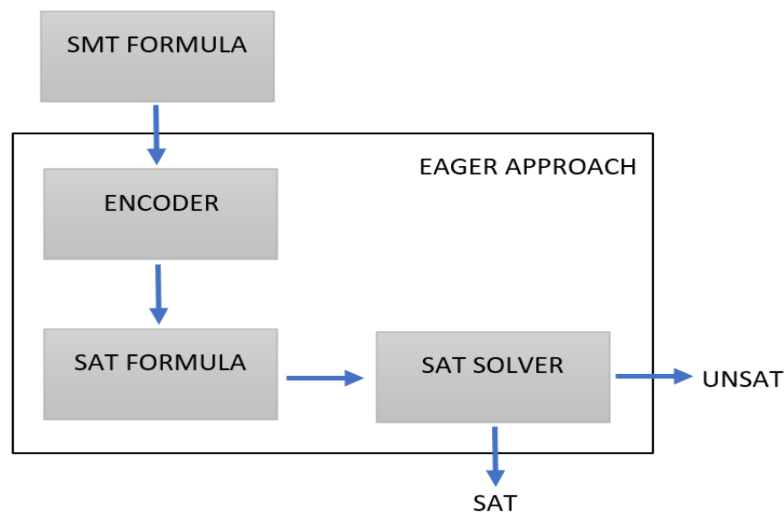

Figure 1.4: Eager approach to SMT [31]

Chapter 4 will provide more detail on the Eager approach. Lazy solvers involve an iterative approach with the combination of SAT solvers and Theory solvers. Theory solvers consist of specialized methods involving data types like strings, Bit Vectors, Arrays, Inequalities, Uninterpreted functions to certain classes of

formulas. SAT techniques are applied to these formulas to determine satisfiability. The Lazy approach is not used in this thesis but a brief overview is provided in Chapter 4 for completeness.

Some solvers favor a specific approach while other may use a combination of both. Boolector, Z3 and CVC4 are all SMT solvers that use the Davis-Putnam-Logemann-Loveland (DPLL) algorithm to determine the satisfiability of Boolean instances where theories have been applied [2, 3, 18]. To create a SMT instance, a Boolean instance is generalized by utilizing different theories and replacing the variables.

Boolector was the chosen SMT solver for this thesis and will be discussed in more detail in Chapters 4 and 5. Chapter 4 will introduce and provide an overview of Boolector. Chapter 5 will present the input format BTOR, the C APIs, usage examples, and the status of LGraph and Boolector integration.

Chapter 2

LIVEHD

This chapter discusses the importance of live hardware development, the open source software tools integrated with LiveHD infrastructure, and the key components of the LiveHD framework.

2.1 Importance of Live Development

Modern technology has conditioned users to expect easy to use products with fast results. Technology improvements that fostered these expectations fuel the development of different methods for processing data. Live processing is important because it increases productivity and keeps the user engaged. This method of processing data involves processing updates with the ability to change code while a program is running. Live processing is different from batch processing where a

job or request is submitted and then a response or result is generated hours later. The change between processing methods extensively shortens the wait time from hours to minutes or less. A decreased wait period is specifically helpful in lengthy processes such as the development and design of new semiconductor chips.

2.2 Open-Source Third Party Tools

LiveHD [27] utilizes integrated third party tools as part of the main infrastructure components. Yosys, ABC, OpenTimer, and Mockturtle provide the latest methods and frameworks for open source synthesis and simulation.

Yosys [30] is an open source framework for register transport level (RTL) synthesis providing algorithms for various application domains with Verilog support. With the ability to create custom synthesis flows, Yosys also has support for non-synthesis applications and creating extensions to Yosys with its integrated C++ APIs.

ABC [16] is a software system that combines scalable logic optimization for sequential synthesis and verification of binary sequential logic circuits in synchronous hardware designs. Data structures have been developed to represent combinational and sequential networks in various ways such as a netlist and And-Inverter-Graphs (AIG) technology-mapped networks.

OpenTimer [28] provides simulation for very large scale static timing analysis

(STA) during IC development. During the design flow, local operations like net rerouting, have the ability to greatly impact the local and overall timing but depending on the change, only a small fraction of the timing may need to be updated. This software verifies the expected timing behaviors using incremental timing for various optimization flows.

Mockturtle [28] is a C++ 17 library that provides generic logic synthesis algorithms and network data structures. The design is based on 4 principle layers dependent on each other with the linear order of network interface API, algorithms, network implementations, and performance tweaks[15]. Mockturtle offers an easily integrated solution for logic network representation and manipulation or compiled and used as a stand-alone logic synthesis tool[14].

Additional open source tools and libraries such as, LiveSim, SMatch, and LiveSynth, have been integrated with LiveHD [21, 28, 27] to provide developers with a robust infrastructure designed for live hardware development. There are many open-source tools that represent different stages of digital design flow; however, using multiple tools comes with drawbacks. Open-source tools often use different data structures and have different standards which make integration difficult. An example is code duplication from each EDA tool's netlist parser which causes additional flow execution time.

2.3 LiveHD Infrastructure

LiveHD is unique in that it aims to integrate several open-source EDA tools under a set of application programming interfaces (APIs) and a single data model [28]. The infrastructure is optimized for synthesis and simulation of ASIC and FPGA designs. The goal of the infrastructure is to represent different stages of the digital design flow and return results in a matter of seconds for small design changes [27]. LiveHD offers a custom shell interface that makes development with the integrated open-source tools easy.

LiveHD is the overarching framework on top of LNAST and LGraph. Language Neutral Abstract Syntax Tree (LNAST), is used to interface between LiveHD and high-level HDLs such as Verilog or Pyrope [29]. The Live Graph, LGraph, in Figure 2.1, is the intermediate representation (IR) of a design database built for live hardware development.

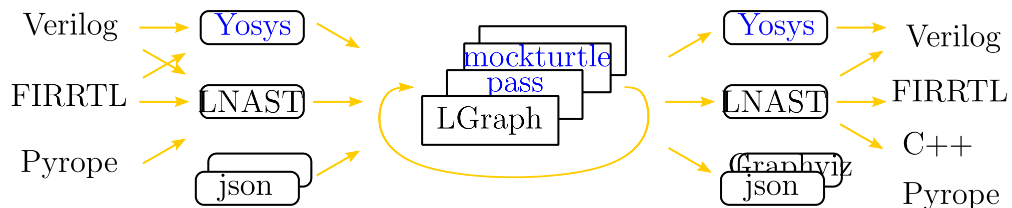

Figure 2.1: LiveHD Infrastructure[27]

To use LiveHD, a developer begins with an RTL description such as Verilog

or Pyrope. The description is read in with the help of LNAST or Yosys using the shell, and a graph structure is generated. An optimization or analysis of the IR, can then be performed using the shell environment before it is read out back to an HDL. The optimization is referred to as a `PASS` which describes transformation over an existing LGraph. For live synthesis, LiveHD uses LiveSynth and SMatch[23, 22]. LiveSim is the live simulation framework used for RTL simulation[10]. This thesis involves work with LGraph.

Chapter 3

LGRAPH

Chapter 3 will discuss the LGraph component of LiveHD in detail and the proposed improvements to the infrastructure. The proposed improvements will outline the main goals of this thesis.

3.1 Overview

LGraph stands for live graph which is a graph representation optimized to represent netlists during different phases of synthesis and physical implementation [21]. A single LGraph represents a single netlist module formed of nodes, node pins, edges and tables of attributes[21, 27]. Complex HDL modules with functions can consist of multiple LGraphs.

3.2 Nodes

A key component of LGraph is the node which represents a vertex in a LGraph graph. The nodes consist of different types that represent logic operations, arithmetic operations, registers, constants, muxes, wire selections and subgraphs.

Every LGraph, node, and pin have a unique pin identifier. Listing 3.1 shows the Verilog code of trivial.v from the LiveHD Github [27]. Trivial.v is a simple XOR circuit with two inputs and one output.

```
module trivial( input a, input b, output c );
assign c = a ^ b;
endmodule
```

Listing 3.1: Verilog Code of trivial.v

The visual LGraph representation of trivial.v, seen in Figure 3.1, was generated using Graphviz. The visual demonstrates the Verilog representation of two inputs and one output along with the identifying information added by the LGraph translation. Each netlist gate is assigned with a node ID and each port of the gate is assigned a port ID (PID)[21, 28, 27].

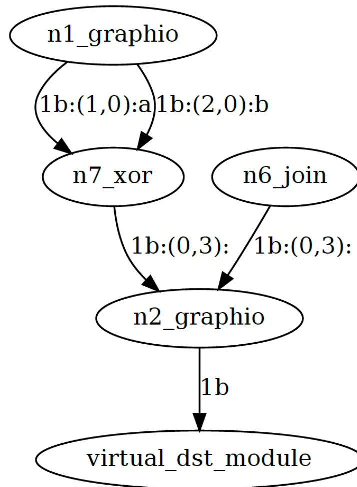

Figure 3.1: LGraph of trivial.v using Graphviz[8]

Each node has input pins and output pins that are labeled as driver or sink pins. The driver pins are outputs of a node and sink pins are the inputs of the node. The directed arrows between nodes have a label that consist of the number of bits with the letter b, followed by the driver and sink pin IDs, in listed order, inside parenthesis, and the driver pin name if given. The graphio nodes indicated the inputs and outputs of the graph. For the graph in Figure 3.1, the two directed arrows from n1_graphio indicated two inputs and the singled directed arrow from n2_graphio indicates a single output to "virtual_dst_module" which is the output label. The XOR is the gate found while traversing and the JOIN node indicates a wire. The labels and nodes of each Graphviz representation of LGraph will vary with each circuit.

3.3 Edges

A LGraph edge defines a wire or collection of wires. The edge indicates a pair of directly connected driver and sink pins. LGraph does not assign unique identifiers to edges[21, 28].

3.4 Graph Traversals

The graph is bidirectional and supports topological and hierarchical traversals. The topological traversals are performed in an input-forward and output-backward manner[28]. Different types of iterators are used for traversing a LGraph. A forward iterator, seen in Listing 3.2, indicates a particular visitation order in which all constant labeled nodes are visited first. The forward traversal does not traverse subgraph nodes[27].

```
for(const auto &node : g->forward () )
```

Listing 3.2: Forward traversal code example

The fast iterator, in Listing 3.3, allows visiting all of the nodes at random and does not traverse sub-graph nodes.

```
for(const auto &node : g->fast () )
```

Listing 3.3: Fast traversal code example

A backward traversal, seen in Listing 3.4, uses the backwards iterator and will visit sub-graph nodes recursively.

```
for(const auto &node : g->backward () )
```

Listing 3.4: Backward traversal code example

In a hierarchical traversal, a required top-level graph can consist of many sub graphs. When a sub-graph node is encountered during a hierarchical traversal, a new LGraph is generated with the exception of sub-graph inputs and outputs[27, 28]. All LGraph inputs and outputs have hard coded values. This traversal type walks through the sub-graph contents when encountered. Figure 3.2 provides an example of a hierarchical model.

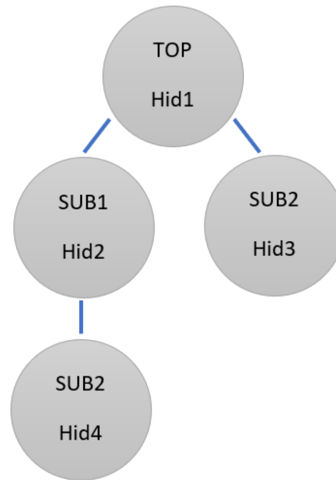

Figure 3.2: Hierarchical traversal example[28]

Sub1 and Sub2 nodes represent sub-graph structures. The extension of the Sub1 node indicates a walk through the sub-graph. Figure 3.3 illustrates the graph example of Figure 3.2.

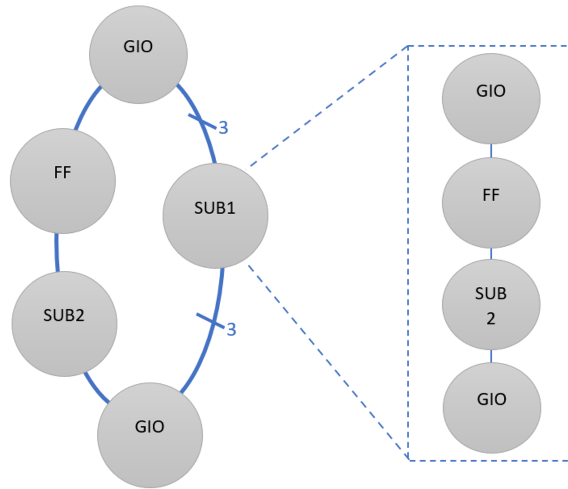

Figure 3.3: Hierarchical Traversal Graph Structure[28]

The top level node is uniquely identified first then the sub-graphs and last, the sub-sub-graph. This cross module traversal treats a hierarchical netlist like a flattened design allowing for better optimization of performance [27, 28].

Chapter 4

SAT AND SMT SOLVERS

This chapter will cover background information on satisfiability solving techniques, their relation to equivalence checking and methods used for the LiveHD LEC pass. Functional verification quickly becomes a bottleneck during the design process with the increased complexity of chip design. Modern equivalence checking software uses a combination of fundamental techniques and evolved algorithms to speed up the design process and reduce time to market.

4.1 SAT Solvers

Boolean satisfiability (SAT) is the problem of determining whether there exists a variable assignment to a Boolean formula such that it evaluates to true. SAT was the first known nondeterministic polynomial-time (NP) complete prob-

lem originally proven by Stephen Cook in 1971 [7]. Algorithms with the worst case time complexity are known for being NP-complete and often have instances involving tens of thousands of variables and millions of constraints referred to as clauses.

Logic gates are expressed in Conjunctive Normal Form (CNF) with their inputs and outputs as Boolean variables. CNF is the conjuction of one or more clauses where each clause is a disjunction of literals. Figure 4.1 shows AND, OR, and XOR logic gates and their respective CNF forms.

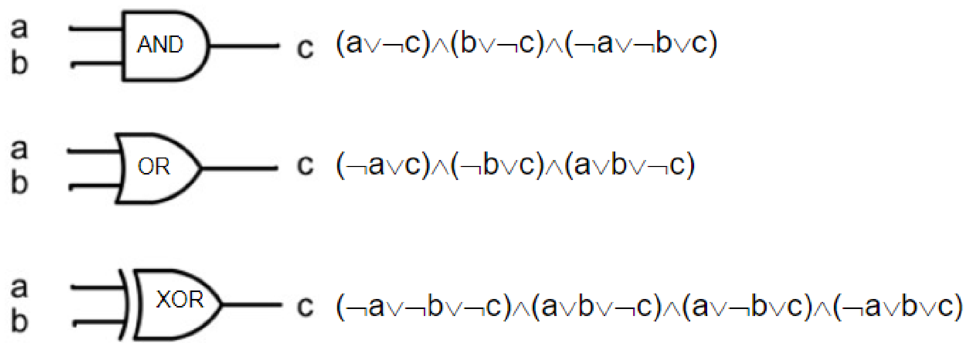

Figure 4.1: Boolean CNF of Logic Gates

As described by Mohnke et al.[17], new and efficient algorithms for SAT have been developed, allowing larger problem instances to be solved and increasing the number of applications in electronic design automation. SAT is now used to solve EDA problems in areas such as test pattern generation, logic optimization, bounded model checking and formal equivalence checking. In equivalence check-

ing, SAT is used to check the functional equivalence between two circuits using a miter circuit. A miter circuit, shown in Figure 4.2, is considered UNSAT, if and only if, the two circuits being compared are equivalent.

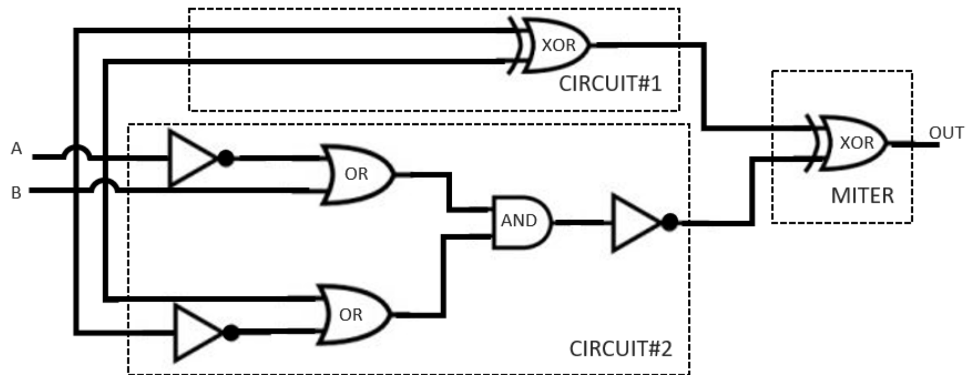

Figure 4.2: Miter Circuit for Logical Equivalence Checking

Figure 4.2 shows Circuit#1 and Circuit#2 having the same a and b inputs. If the output of the miter is 0, the circuits are considered equivalent and the miter circuit will always output 0 when the circuits agree on the given input.

A larger circuit usually indicates a larger CNF representation, but a circuit can be represented by a liner-size CNF encoding using the Tseitin transformation [26]. The disadvantage of using SAT to verify functional equivalence is that SAT is a NP-Complete problem [7], thus a NP-Hard problem, meaning a solution may not be decidable [13].

4.2 SMT Solvers

Satisfiability Modulo Theories (SMT) refers to the problem of deciding the satisfiability of a first-order formula by incorporating different supporting theories. SMT solvers are found in hardware and software verification as back-end applications combining multiple theories. Additional theories such as bit vectors, arrays, lists, inequalities, uninterpreted functions and arithmetic are needed as SAT is not ideal for scaling with the amount of components in current integrated circuits. These solvers are important as they allow reasoning over a given domain and provide decidable satisfiability problems for first-order theories and fragments of theories.

SMT solver formulas allow for a more affluent modeling language over SAT solver formulas. For example, a SAT formula represents the data path operations at the bit level of a microprocessor and an SMT formula would represent the data path operations at the word level. The SMT solving strategies consist of converting a SMT formula into a propositional formula that is equisatisfiable. The SMT formula is considered SAT if and only if the SAT formula is considered SAT. Both approaches to SMT solving, Eager and Lazy, follow this rule to determine the satisfiability of a SMT formula.

4.2.1 Eager Approach to SMT Solving

The Eager approach to SMT solving involves encoding all the necessary properties of background theories into a SAT problem before the propositional formula is passed to a SAT solver to determine satisfiability. While converting to between SMT and SAT formulas, the theories being used in SMT are also taken into account. Figure 4.3 provides more detail of the Eager approach to SMT.

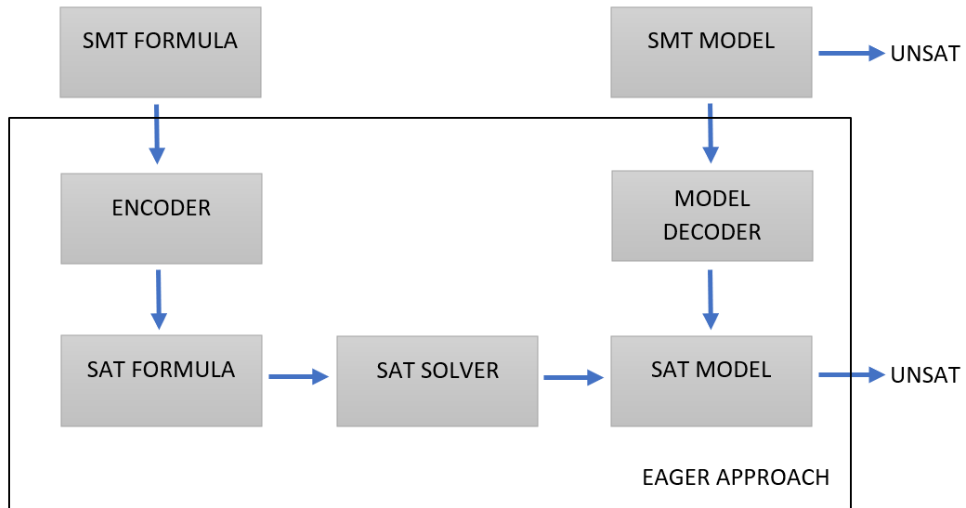

Figure 4.3: Eager approach to SMT[12]

The encoding process commonly involves two schemes, Ackermann and Small Domain encoding. Ackermann encoding eliminates the uninterpreted functions and all applications of uninterpreted functions and replaces them with a symbolic constant. Small Domain encoding is based on a theorem that implies a enumer-

ative approach to finding a SAT assignment. The theorem states, given a SMT formula in a specified theory, the SMT formula is considered SAT if and only if there exists a SAT solution to a given SMT formula with all domains for function symbols having cardinality bounded by a finite integer.

4.2.2 Lazy Approach to SMT Solving

As previously mentioned, the lazy approach involves a combination of SAT solvers and Theory solvers using an iterative approach. The theories are specialized methods that restrict their language to certain classes of formulas. SAT techniques are applied to these formulas to determine satisfiability. Figure 4.4 illustrates how SAT and theory solvers are utilized in the Lazy approach to SMT solving.

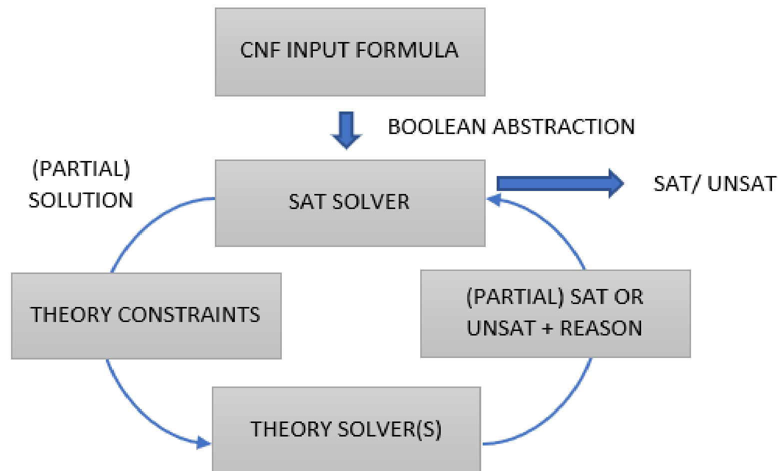

Figure 4.4: Lazy approach to SMT [31]

Lazy solver treat the input formula as being propositional until they are forced to treat it otherwise. During this process, if the SAT solver determines the propositional formula is UNSAT, then the SMT formula is declared UNSAT.

4.3 Boolector

Developed by a group of individuals from Johannes Kepler University Linz, Boolector is publicly hosted on GitHub[6]. It has been entered in numerous SAT and SMT competitions from early years in development. The Boolector website hosts news, API documentation, publications, slides, and information on the developers past and present [19]. There are also a number of contributions from

third parties listed on the Github and Boolector websites.

Boolector is an SMT solver that can be used as a standalone solver or as a backend with interfaces for C and Python APIs. This solver is for the theory of fixed-size bit vectors with arrays and uninterpreted functions and any combinations thereof[19].

Chapter 5

BOOLECTOR INTEGRATION

This chapter will cover the methodology of integrating Boolector with LiveHD. The steps of integration are as follows:

1. Learn LGraph structure, attributes, and their respective usage.

2. Learn Boolector APIs and it's usage.

3. Develop LGraph and Boolector connectivity.

Last, the Boolector and LGraph integration will be tested using two circuits and the results will be presented.

5.1 LGraph

Learning LGraph was separated into three steps. The first task involved understanding node types and edges. Next, was to identify inputs and outputs of an LGraph. The last step involved understanding node connectivity.

5.1.1 Node Types and Edges

Determining the attributes and node type are essential to translate the circuit into a form solvable by SAT or SMT solvers. The identification of each node would need to occur during a traversal.

The `get_name()` attribute identifies the current node name and `get_nid()` defines each node's unique position within the LGraph. These node attributes were used directly after the forward traversal statement to identify the current node being visited.

```
for( const auto &node : g -> forward() )
{
    fmt::print("Node:{} ID:{}\n", node.get_name(), node.
   get_nid());
}
```

Listing 5.1: Code to determine current node

Listing 5.1 demonstrates the code for printing the debug statements. The response of node attributes can be returned as Boolean, string, or numbered responses. Querying if a node is a root node is an example of a Boolean response. The debug name of a node is a case for a string response. The number of input or output edges are examples of attributes with a numbered response. Throughout this task, the additional node attributes listed in Table 5.1 were used. Not all existing attributes are listed.

Node Attribute	Return Description
Node::has_inputs()	Boolean, if node as inputs
Node::has_outputs()	Boolean, if node as outputs
Node::get_name()	Name of node
Node::get_num_edges()	Number of edges
Node::get_num_inp_edges()	Number of input edges
Node::get_num_out_edges()	Number of output edges
Node::get_bits()	Number of bits
Node::get_place()	Location in LGraph
Node::get_debug_name()	Debug name of node
Node::is_root()	Boolean, if node is root node
Node::get_type_op()	Node operation type

Table 5.1: LGraph Node Attributes

The name and debug name of a node are strings that can be changed. Thus, the node type operation, `NType_op`, must also be verified. Not all node type operations are listed in Table 5.2.

Node Type Operation	Description
NType_op::And	Outputs logical AND of inputs
NType_op::Or	Outputs logical OR of inputs
NType_op::Xor	Outputs logical XOR of inputs
NType_op::Ror	Outputs logical Reduce OR of inputs
NType_op::Not	Outputs logical negation of inputs
NType_op::LT	Less Than
NType_op::GT	Greater Than
NType_op::EQ	Equal
NType_op::SHL	Logical Shift Left by given bits
NType_op::SRA	Logical Shift Right by given bits
NType_op::IO	Graph Input or Output
NType_op::Const	Constant
NType_op::Sum	Output sum of inputs
NType_op::Mult	Output product of inputs
NType_op::Div	Output division of inputs

Table 5.2: LGraph Node Type Operations[27]

Combining the usage of node type operations, `NType_op`, and the `Node` attributes, the node type operation can be found while traversing a LGraph. When a specific node type is found, additional operations can be performed based on the `NType_op`. Placing the code, as showing in Listing 5.2, inside the LGraph traversal loop, each node operation will also be identified during the same node visitation.

```
if(node.get_type_op() == NType_op::And)
{
    //Insert operations to be performed
}
```

Listing 5.2: Combining Node and Node_pin Attributes

An edge was previously defined as a connection between nodes. Since LGraph allows for forward and reverse traversals, all edges are considered bidirectional. The most important edge attribute for this thesis was `get_bits()` which returned the number of driver bits translating to the input bit width of a node.

5.1.2 Identifying LGraph Inputs and Outputs

Traversals were performed according to the method outlined in Section 3.4. Using the different traversal methods, the inputs and outputs were identified for each LGraph according to Listing 5.3 and Listing 5.4.

```
g->each_graph_input([&](const Node_pin &input_pin)
{
    fmt::print("LGraph input: {} {}\n", input_pin.get_name()
   , input_pin.get_pid());
}
```

Listing 5.3: Code to generate LGraph inputs

`Node_pin`, is used to determine the inputs and outputs of a node. A node pin consists of an index that indicates the node the pin is connected to. The node pin also has a port ID, which identifies the node within the LGraph being traversed. The code for iterating over the inputs and outputs is the same except for the specified node pin; input or output. The pin name, `get_name()`, and pin id ,`get_pid()`, attributes are used in the debug print statement.

```
g->each_graph_input([&](const Node_pin &output_pin)
{
    fmt::print("LGraph output: {} {}\n", output_pin.get_name
   (), output_pin.get_pid());
}
```

Listing 5.4: Code to generate LGraph outputs

Table 5.3 offers additional attributes and their description for `Node_pin` usage.

Not all possible attributes are listed. Depending on the attribute, the return can be a Boolean, string, or numbered responses. A Boolean response is given for an inquiring attribute such as determining if the node pin has outputs. A string usually provides a name. Retrieving IDs and getting the count of an attribute are provided with a numbered response.

Node_Pin Attribute	Return Description
Node_Pin::has_inputs()	Boolean, if node pin has inputs
Node_Pin::has_outputs()	Boolean, if node pin as outputs
Node_Pin::is_graph_io()	Boolean, if node pin is IO type
Node_Pin::is_graph_input()	Boolean, if node pin is input
Node_Pin::is_graph_output()	Boolean, if node pin is output
Node_Pin::is_type_const()	Boolean, if node pin is constant
Node_Pin::get_node()	Associated node
Node_Pin::get_node_nid()	Unique ID in LGraph
Node_Pin::get_type_op()	Node operation type
Node_Pin::get_driver_node()	Connected driver node
Node_Pin::get_driver_pin()	Connected driver node pin
Node_Pin::get_bits()	Number of bits in node pin

Table 5.3: LGraph Node_pin Attributes

5.2 Boolector

Fundamentally, Boolector is a SMT solver that supports the Satisfiability Modulo Theories Library, SMT-LIB v2, and BTOR logics. Designed with the intention of creating a common standard for comparing SMT systems, SMT-LIB is continuously evolving and adding to its library of benchmarks[1]. BTOR is the bit-precise, word-level format for formulas over bit-vectors in combination with one-dimensional arrays[5].

The rich C API provided allows developers to use Boolector as a library with the logic depicted in Figure 5.1. The Parser reads SMT-LIB or BTOR inputs and builds a directed acyclic graph (DAG). The DAG is simplified by basic rewriting rules. The Rewriter provides rules that are divided into basic rules applied to during formula construction, global term substitutions and static analysis techniques, and arithmetic normalization. Formula Refinement calls the SAT solver and returns a SAT solution if found, or directs an UNSAT to the different refinement methods to search for a SAT solution. The Array Consistency Checker determines of the current SAT assignment is consistent with the theory of arrays. Under-Approximation is used for adding constraints as clauses to the CNF. The Model Generator provides SAT models when enabled[18].

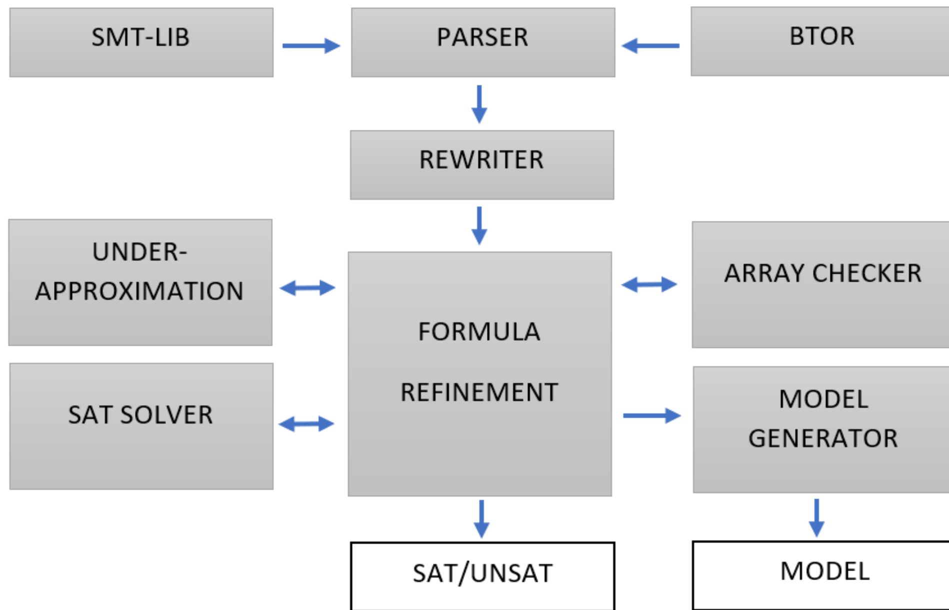

Figure 5.1: Boolector Logic Flow Chart [4]

As a SAT solver, Boolector's back end supports CryptoMiniSat, Lingeling, PicoSAT, MiniSAT and CaDiCaL [6]. Boolector allows the end user to choose the SAT solver during code development. As of the 2019 SMT competition, CaDical is the default SAT engine for Boolector[20]. Depending on the SAT solver chosen, some functionality may be limited.

5.2.1 BTOR Format

The BTOR logic format was created by the developers of Boolector to solve issues with alternative input formats. Existing input formats did not adequately handle model checking problems and had restrictive, complex, and error prone interpretations. BTOR offered the solution by creating a simple, direct and standardized input format.

BTOR supports bit-vector variables, constants, and one-dimensional bit-vector arrays with an arbitrary length based off principles from bit-level AIGER format and SMT-LIB logics. Boolean variables are treated as bit-vectors with bit-width one to eliminate unnecessary conversions. Plus, modeling sequential and synchronous circuits are also BTOR supported[5].

5.2.2 C APIs

Boolector C APIs are built using the BTOR format operators consisting of macros and functions. First, a Boolector instance must be generated using code in Listing 5.5.

```
BTOR *btor = boolector_new();
```

Listing 5.5: Code to create Boolector Instance

To clone a Boolector instance, the instance is used as an input parameter. The

instance can be cloned at anytime if the Lingleing SAT solver is set. Otherwise, a clone must be called, as in Listing 5.6, before determining if a Boolector instance is SAT or UNSAT.

```
BTOR *boolector_clone(Btor *btor);
```

Listing 5.6: Code to clone a Boolector Instance

Internally, Boolector maintains the directed acyclic graph of expressions that are simplified and processed shown in Figure 5.2. The instance is then entered as an input parameter in BoolectorNode APIs representing an expression or operation in the DAG as a vertex.

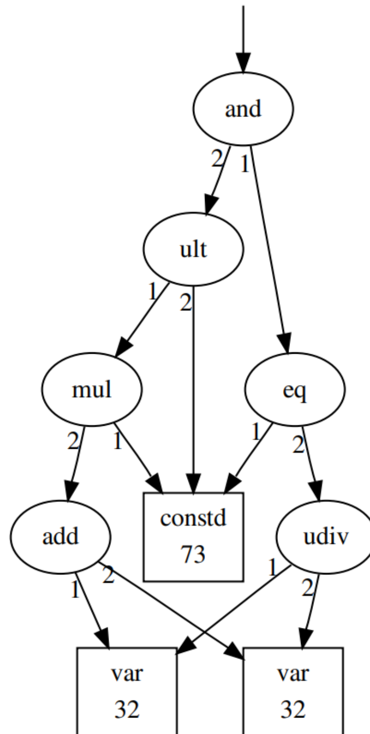

Figure 5.2: Example of Directed Acyclic Graph (DAG)

The Boolector instance and BoolectorNode are taken as parameters when asserting or releasing an expression as shown in Listing 5.7. Adding an assertion will add an expression to the formula permanently and increment the reference counter until the instance is deleted entirely, the expression is released from memory, or overwritten. Releasing an expression from memory decrements the reference counter.

```
boolector_assert(Btor *btor, BoolectorNode *n0);
```

```
boolector_release(Btor *btor, BoolectorNode *n0);
```

Listing 5.7: Code to assert or release BoolectorNode

To delete a Boolector instance, Listing 5.8 shows the instance is taken as parameter[6]. Before deleting the instance, the reference counter should be zero otherwise a portion of the instance will remain in memory.

```
boolector_delete(Btor *btor);
```

Listing 5.8: Code to delete Boolector Instance

BoolectorNodes consist of bitwise, boolean, arithemetic, relational, shifting and negation reduction arithmetic classes[5]. All nodes will take the previously declared Boolector instance as a parameter along with one or two BoolectorNodes depending on the operator. In Table 5.4 the Boolector instance is referred to as *btor and the BoolectorNodes are referred to as *n0 and *n1.

Boolector Node	Operator Description
boolector_and(*btor,*n0,*n1)	Bit-vector AND
boolector_or(*btor,*n0,*n1)	Bit-vector OR
boolector_xor(*btor,*n0,*n1)	Bit-vector XOR
boolector_ror(*btor,*n0,*n1)	Rotate Right
boolector_not(*btor,*n0)	One's complement of *n0
boolector_slt(*btor,*n0,*n1)	Signed less than
boolector_sgt(*btor,*n0,*n1)	Signed greater than
boolector_eq(*btor,*n0,*n1)	Bit-vector or array equality
boolector_sll(*btor,*n0,*n1)	Logical shift left
boolector_srl(*btor,*n0,*n1)	Logical shift right
boolector_const(*btor,*bits)	Bit-vector constant of *bits
boolector_add(*btor,*n0,*n1)	Bit-vector addition
boolector_mul(*btor,*n0,*n1)	Bit-vector multiplication
boolector_sdiv(*btor,*n0,*n1)	Signed division

Table 5.4: Boolector Node Types

The description of Boolector nodes listed, are easily correlated to the LGraph node type operations. For this thesis, the BoolectorNodes and LGraph node types of Table 5.5 are defined as operation equivalent.

LGraph NodeType Operation	**Boolector Node**
NType_op::And	boolector_and(*btor, *n0, *n1)
NType_op::Or	boolector_or(*btor, *n0, *n1)
NType_op::Xor	boolector_xor(*btor, *n0, *n1)
NType_op::Ror	boolector_ror(*btor, *n0, *n1)
NType_op::Not	boolector_not(*btor, *n0)
NType_op::LT	boolector_slt(*btor, *n0, *n1)
NType_op::GT	boolector_sgt(*btor, *n0, *n1)
NType_op::EQ	boolector_eq(*btor, *n0, *n1)
NType_op::SHL	boolector_sll(*btor, *n0, *n1)
NType_op::SRA	boolector_srl(*btor, *n0, *n1)
NType_op::Const	boolector_const(*btor, *bits)
NType_op::Sum	boolector_add(*btor, *n0, *n1)
NType_op::Mult	boolector_mul(*btor, *n0, *n1)
NType_op::Div	boolector_sdiv(*btor, *n0, *n1)

Table 5.5: LGraph and Boolector Node Type Equivalents

To determine if a Boolector instance is SAT or UNSAT, the default solver can be used. Listing 5.9 and Table 5.6 demonstrate how a specific SAT solver can be set using the Boolector instance and SAT typedefs as input parameters for more control.

```
boolector_set_sat_solver(Btor *btor, const char *solver);
```

Listing 5.9: Code to set Boolector SAT Solver

Boolector SAT Typedefs	SAT Engine
BTOR_USE_LINGELING	Lingeling
BTOR_USE_CADICAL	CaDiCaL
BTOR_USE_MINISAT	MiniSAT
BTOR_USE_PICOSAT	PicoSAT
BTOR_USE_CMS	CryptoMiniSat

Table 5.6: Boolector SAT Engine Typedefs[6]

The simplify function is called to apply additional simplification at the specified instance to the current input formula. Listing 5.10 shows the Boolector instance as the only input parameter.

```
boolector_simplify(Btor *btor);
```

Listing 5.10: Code to simplify Boolector input

To determine satisfiability of the input formula, the Boolector instance is entered as the parameter to the SAT function. This function may only be called once if the Incremental usage mode is not enabled.

```
boolector_sat(Btor *btor);
```

Listing 5.11: Code to call Boolector SAT

The SAT call returns the macros for SAT, UNSAT, or UNKNOWN. The macros are preprocessor constants that indicate the status. The input formula is represented by the Boolector instance defined by the assertions from the code in Listing 5.7 and 5.12.

```
boolector_assert(Btor *btor, BoolectorNode *n0);
```

Listing 5.12: Code to simplify Boolector input

When incremental usage is enabled, assumptions can be made to guide the search for a solution. Assertions and assumptions are combined using the Boolean AND.

5.2.3 Example Usage

This section will provide usage examples of the Boolector APIs utilized before integrating them with LGraph APIs. After a Boolector instance is generated, a data type must be defined before Boolector variables, `vars`, can be created. Listing 5.13 presents how BoolectorSort defines `bsort16` as the datatype equivalent to a vector of 16 bits assigned to the Boolector instance `btor`.

```
Btor *btor = boolector_new();
BoolectorSort bsort16 = boolector_bitvec_sort(btor, 16);
```

Listing 5.13: Define Boolector *btor and data type Example

Listing 5.14 depicts how Boolector expressions are defined by the Boolector instance, a size defined by BoolectorSort, and a symbol as input parameters. The symbol must be a unique string or set to NULL if no symbol is assigned.

```
BoolectorNode *x = boolector_var(btor,sort,"xName");
BoolectorNode *tempNode = boolector_and(btor,in1,in2);
```

Listing 5.14: BoolectorNode Usage Example

BoolectorNodes are made from variables and operation types as shown in Listings 5.14 and 5.15. Each node and associated inputs are added to the Boolector instance during traversal via the code outlined in Listing 5.15.

```
formula = boolector_xor(btor,x1,x1);
boolector_assert(btor,formula);
```

Listing 5.15: Boolector Formula Assertion Example

Upon finishing traversal, the finalized formula is asserted and a call for to SAT is performed. Finally, all data and nodes types must be released be release from memory before a Boolector instance can be cleared from memory. This process is shown in Listing 5.16.

```
boolector_release(btor,x)
boolector_release_sort(btor,bsort16)
boolector_delete(btor);
```

Listing 5.16: Release Boolector Usages

5.3 Integration of Boolector and LGraph

For this thesis, the Boolector API and usage examples were used to develop the final version of "PASS.LEC", the logic equivalence checking pass for LGraph. A summary and examples from the original source code are provided as follows.

5.3.1 Creating Pass.LEC

Listing 5.17 demonstrates how the pass creates a Boolector instance and defines the input datatype. Two options for model generation and automatic cleanup are set using `boolector_set_opt()`. These options output a solution model for a SAT

solution if found and clears any unreleased memory associated with the Boolector instance.

```
Btor *btor = boolector_new();
BoolectorSort s = boolector_bitvec_sort(btor,1);
//Can be removed if model generation unneeded.
boolector_set_opt(btor,BTOR_OPT_MODEL_GEN,1);
boolector_set_opt(btor,BTOR_OPT_AUTO_CLEANUP,1);
```

Listing 5.17: Setup Boolector for PASS.LEC

The forward traversal of an LGraph differs with each input circuit. It consists of identifying the number of input bits, node, and node type operation. During traversal, PASS.LEC prints out each node and the node ID. Depending on the node operation type, additional operations take place between identifying the LGraph node type operation and BoolectorNode. These operations allow the LGraph nodes to conform to the input parameters of the matching BoolectorNode operation. The BoolectorNode is assigned to a temporary node to eliminate the need for declaring all node types. The temporary BoolectorNode is asserted using the previously mentioned input parameters and the memory for the temporary node is cleared.

The last assertion before calling SAT, is the formula assertion. The number of

graph inputs are assumed to be 2, for now. The miter circuit is created using the Boolector XOR node and defined inputs to test the circuit's satisfiability. The x1 and x2 inputs for the XOR node are used as place holders in the code example below in Listing 5.18.

```
formula = boolector_xor(btor,x1,x2);
boolector_assert(btor,formula);
int result = 0; //ensure result is cleared
result = boolector_sat(btor);
```

Listing 5.18: Boolector Formula Assertion Example

Listing 5.19 shows how the result returns and prints out "SAT" or "UNSAT" if the value of result is equivalent to BOOLECTOR_SAT or BOOLECTOR_UNSAT defines. If result returns any other integer, "UNKNOWN" is output. A model is generated if a SAT result if found. Model generation is enabled in PASS.LEC, otherwise no model solution will be given. For larger circuits, model generation can be disabled.

```
if(result == BOOLECTOR_SAT)
{
    fmt::print("Result: SAT\n");
    char* b = (char*)"btor";
    boolector_print_model(btor,b,stdout);
}
else if(result == BOOLECTOR_UNSAT)
{ fmt::print("Result: UNSAT\n"); }
else
{ fmt::print("Result: UNKNOWN\n");}
```

Listing 5.19: Code for returning solution to SAT call

The end of the pass involves freeing each BoolectorNode from memory using `boolector_release()`. All nodes must be released before the data type and instance can be freed. Listing 5.20 shows how the data type is released using `boolector_release_sort()` and comes before deletion of the Boolector instance using `boolector_delete()`.

```
boolector_release(btor,x)

boolector_release_sort(btor,s)

boolector_delete(btor);
```

Listing 5.20: Releasing Boolector usages from PASS.LEC

5.3.2 Testing PASS.LEC

To test the LEC pass, the LiveHD shell was used to read in two or more Verilog files. The `inou.yosys.tolg` command reads in a Verilog file using Yosys and translates the file to an LGraph structure. The command must be paired with the file location. The following inputs to the shell, in Listing 5.21, are used to read in trivial.v and trivial3.v Verilog files.

```
livehd> inou.yosys.tolg files:inou/.../trivial.v


livehd> inou.yosys.tolg files:inou/.../trivial3.v
```

Listing 5.21: Commands to generate LGraphs trivial and trivial3

To open each LGraph, the `lgraph.match` command is used. Listing 5.22 demonstrates how `lgraph.match` and `pass.lec` commands are used to check if

all LGraph outputs are satisfiable.

```
lgraph.match |> pass.lec
```

Listing 5.22: Command to test LEC pass

The pass outputs the number of LGraphs read in, the name of each LGraph, and whether each graph is SAT or UNSAT. If a LGraph is found to be SAT, a valid model is provided. The number of references to Boolector is printed to ensure there are no remaining instances of Boolector left in memory.

```
inou.yosys.tolg module.trivial3
lgraph.match |> pass.lec


Starting PASS.LEC
Number of LGraphs: 2
---INPUTS OF: trivial
NAME: a PID:1 BITS:1
NAME: a PID:2 BITS:1


---TRAVERSING: trivial
Node Type: const, place: 9
Node Type: get_mask, place: 10
```

```
Node Type: get_mask, place: 7

Node Type: xor, place: 6

 xor found at 6

sat

2 1 in1

3 0 in2

Boolector Refs:0


---INPUTS OF: trivial3

NAME: a[0] PID:1 BITS:1

NAME: a[1] PID:2 BITS:1


---TRAVERSING: trivial3

Node Type: const, place: 9

Node Type: get_mask, place: 10

Node Type: get_mask, place: 7

Node Type: xor, place: 6

 xor found at 6

sat

2 1 in1

3 0 in2
```

```
Boolector Refs:0


PASS.LEC COMPLETED
```

Listing 5.23: Output from PASS.LEC

The current implementation only assumes two inputs with the expansion of more inputs in progress. For the two assumed inputs, larger input bit width can be handled. The process involves checking input bit width first, then declaring bit width size as the size of the Boolector vector for the BoolectorSort data type.

The current status of the LEC pass provides primary connectivity between LGraph and Boolector. The pass can determine node type and operations, location, inputs, outputs, sink and driver pin information, bit width, and naming of LGraph nodes. Required node types have been translated between Boolector C APIs and the LGraph framework. The translation involves conforming to the LGraph data types and requirements after Boolector optimizations and operations have occurred. The LEC pass checks the satisfiability of each LGraph for combinational circuits and provides a SAT model if found.

Larger designs involving sequential circuits are not handled gracefully with the current code base. The logic for handling sequential circuits is under development and thus will exit when encountered during traversal.

Chapter 6

CONCLUSION AND FUTURE WORK

This thesis presented the foundational work for developing a Logic Equivalence Checking pass for use with LiveHD. The LiveHD graph structure, LGraph, and an SMT solver were utilized to ensure equivalence between optimizations while working with any combination between netlists or netlists and hardware descriptive languages. Boolector was the chosen SMT solver for integration with the LiveHD framework. The primary goal of creating a LEC pass was to eliminate the need of external logic checkers, such as Formality, by integrating a SMT solver and provide an open-source option for this open-source EDA tool.

The current LEC pass has the limitations of assuming two inputs and only

handling combinational logic. However, required node operations were successfully translated between LGraph and Boolector APIs. A SAT model can be provided for combinational circuits with added sequential circuit capability in progress. As it is still under development, PASS.LEC does not fully pass the LiveHD regression testing due to the inability to handle sequential circuits and the limitation of two assumed inputs.

Future work for PASS.LEC involves the implementation of LEC for all variations of combinational and sequential circuits. The expansion of more that two inputs must be handled as well as aiming to pass regression testing. These capabilities would allow LiveHD, for example, to check two or more larger LGraph representations against each other to determine satisfiability and minimize the dependence on external checkers like Formality.

Bibliography

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib), 2016.

[2] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[3] Nikolaj Bjorner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. The z3 theorem prover.

[4] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. pages 174–177, 03 2009.

[5] Robert Brummayer, Armin Biere, and Florian Lonsing. Btor: bit-precise modelling of word-level problems for model checking. pages 33–38, 07 2008.

[6] Robert Brummayer, Armin Biere, Aina Niemetz, and Mathias Preiner. Boolector's api documentation, 2016.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[8] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003.

[9] Evgueni Goldberg, Mukul Prasad, and Robert Brayton. Using sat for combinational equivalence checking. pages 114–121, 02 2001.

[10] S. Hassani, G. Southern, and J. Renau. LiveSim: going live with microarchitecture simulation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 606–617, March 2016.

[11] Shi-Yu Huang and Kwang-Ting(Tim) Cheng. *Formal Equivalence Checking and Design Debugging*. Springer US, 1998.

[12] Tommi Junttila, 2018.

[13] D. E. Knuth. Postscript about np-hard problems. *SIGACT News*, 6(2):15–16, apr 1974.

[14] Siang-Yun (Sonia) Lee and Heinz Riener. Logic synthesis and digital design.

[15] Siang-Yun (Sonia) Lee and Heinz Riener. Mockturtle documentation.

[16] Alan Mischenko. Berkeley logic synthesis and verification group, abc: A system for sequential synthesis and verification.

[17] P. Molitor and J. Mohnke. *Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods*. Springer US, 2013.

[18] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[19] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[20] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector at the smt competition 2019. 2019.

[21] Rafael T. Possignolo, Sheng H. Wang, Haven Skinner, and Jose Renau.

LGraph: A multilanguage open-source database. In *Open-Source EDA Technology, Proceedings of the First Workshop on*, WOSET'18, October 2018.

[22] Rafael Trapani Possignolo and Jose Renau. Livesynth: Towards an interactive synthesis flow. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 74. ACM, 2017.

[23] Rafael Trapani Possignolo and Jose Renau. Smatch: Structural matching for fast resynthesis in fpgas. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 75. ACM, 2019.

[24] Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2015.

[25] Synopsys. *Formality Equivalence Checking and Interactive ECO*. Datasheet.

[26] G. S. Tseitin. On the complexity of derivation in propositional calculus. 1983.

[27] MASC UCSC. Livehd: Live hardware development.

[28] Sheng-Hong Wang, Rafael Trapani Possignolo, Qian Chen, Rohan Ganpati, and Jose 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*, WOSET'19, November 2019.

[29] Sheng-Hong Wang, Akash Sridhar, and Jose Renau. Lnast: A language neutral intermediate representation for hardware description languages. In *Open-Source EDA Technology, Proceedings of the Second Workshop on*, WOSET'19, November 2019.

[30] Clifford Wolf. Yosys Open SYnthesis Suite.

[31] Erika Ábrahám, John Abbott, Bernd Becker, Anna Bigatti, Martin Brain, Bruno Buchberger, Alessandro Cimatti, James Davenport, Matthew England, Pascal Fontaine, Stephen Forrest, Alberto Griggio, Daniel Kroening, Werner Seiler, and Thomas Sturm. Sc^2: Satisfiability checking meets symbolic computation. volume 9791, pages 28–43, 07 2016.