SpyDrNet - An Open-Source Python Netlist Representation for Analysis and Transformation

Dallin Mark Skouson
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/9446

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
SpyDrNet - An Open-Source Python Netlist Representation for
Analysis and Transformation

Dallin Mark Skouson

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Master of Science

Mike Wirthlin, Chair
Brent E. Nelson
Jeff Goeders

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2022 Dallin Mark Skouson
All Rights Reserved
ABSTRACT

SpyDrNet - An Open-Source Python Netlist Representation for Analysis and Transformation

Dallin Mark Skouson
Department of Electrical and Computer Engineering, BYU
Master of Science

SpyDrNet is an open-source structural netlist representation framework written in Python that allows users to create, analyze, and manipulate structural netlists. The internal format was designed to hold generic source structural netlists. Verilog and EDIF netlists generated by Vivado were read and written by SpyDrNet. Additional API functionality was built around the netlist representation. An application applying triple modular redundancy and duplication with compare was created and successfully used.

Keywords: netlists, EDIF, Verilog, SpyDrNet, computer aided design, automation, digital circuits, circuit representations
ACKNOWLEDGMENTS

The efforts of several contributors including Andrew Keller, Jordi Ramos Chen, Michael Bjerregaard, Jacob Brown, and Benjamin Glines among others have been crucial to help make the SpyDrNet tools and this work a possibility. I appreciate their countless hours of work and insightful perspectives.

I would also like to express my gratitude to my mentor and advisor Dr. Mike Wirthlin who inspired and guided my efforts to help shape them into SpyDrNet.

I am grateful to my parents Mark, and Laura, and my siblings, Kate, Ryan, Craig and Mckenna for their encouragement and help, without which I could not have arrived at this work.

This work was supported by the NSF Center for Space, High-Performance, and Resilient Computing (SHREC), supported by the I/UCRC Program of the National Science Foundation under Grant No. 1738550. The opinions, findings, and conclusions or recommendations expressed are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
# TABLE OF CONTENTS

Title Page ................................................................. i

Abstract ................................................................. ii

Acknowledgements ...................................................... iii

Table of Contents ....................................................... iv

List of Tables .......................................................... vii

List of Figures ........................................................ viii

## Chapter 1  Introduction ............................................. 1
1.1 SpyDrNet: Direct Netlist Modifications .......................... 1
1.2 SpyDrNet Features ................................................. 2
   1.2.1 Open Source Nature of the Tools .......................... 3
   1.2.2 Leveraging the Python Language ........................... 3
   1.2.3 Additional Features ...................................... 4
1.3 Contributions of this Thesis .................................... 4

## Chapter 2  Background ............................................. 8
2.1 Netlists ............................................................ 8
   2.1.1 Nets and Components .................................... 8
   2.1.2 Hierarchy in Designs ..................................... 10
2.2 Netlist Text Representations ................................... 11
   2.2.1 Verilog .................................................. 12
   2.2.2 EDIF ............................................... 12
   2.2.3 BLIF ............................................. 13
2.3 Netlist Modification Software .................................. 14
   2.3.1 Verific ............................................. 14
   2.3.2 LiveHD ............................................ 14
   2.3.3 BYU EDIF Tools ..................................... 15
   2.3.4 SpyDrNet: Netlist Representation, Manipulation, and Analysis Framework 15
2.4 Subsequent Chapters ............................................. 16

## Chapter 3  SpyDrNet and its Intermediate Representation .......... 18
3.1 Intermediate Representation Terminology ....................... 18
   3.1.1 SpyDrNet’s Netlist Class ................................ 19
   3.1.2 Cables and Wires ....................................... 21
   3.1.3 Definitions and Instances .............................. 21
   3.1.4 Ports and Pins ....................................... 22
3.2 Object Relationships .............................................. 22
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Metadata for a Generic Representation</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Independence of the Intermediate Representation</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Mutability</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Netlist Connectivity for Analysis</td>
<td>26</td>
</tr>
<tr>
<td>3.7</td>
<td>API</td>
<td>28</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Netlist Creation and Modification</td>
<td>28</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Clone</td>
<td>29</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Getter functions</td>
<td>30</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Hierarchical References</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Parsing and Composing the EDIF Format</td>
<td>33</td>
</tr>
<tr>
<td>4.1.1</td>
<td>EDIF Rename Construct</td>
<td>33</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Single Bit Nets</td>
<td>34</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Ordered Cell Definitions</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Parsing and Composing the Verilog Format</td>
<td>35</td>
</tr>
<tr>
<td>4.2.1</td>
<td>The Assignment Statement</td>
<td>35</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Module Port Declarations</td>
<td>36</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Port Concatenations</td>
<td>36</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Parsing Instances Before Definitions</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Callback Framework</td>
<td>39</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Creating a Plugin</td>
<td>39</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Interface with the API</td>
<td>40</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Watched features</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Benefits of Leveraging the Callback Framework for Namespace Managers</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Namespaces</td>
<td>41</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Organization</td>
<td>42</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Manager</td>
<td>43</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Using the namespace managers</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Flatten</td>
<td>46</td>
</tr>
<tr>
<td>6.2</td>
<td>Uniquify</td>
<td>47</td>
</tr>
<tr>
<td>6.3</td>
<td>Redundancy and Reliability</td>
<td>48</td>
</tr>
<tr>
<td>6.3.1</td>
<td>TMR</td>
<td>48</td>
</tr>
<tr>
<td>6.3.2</td>
<td>DWC</td>
<td>49</td>
</tr>
<tr>
<td>6.4</td>
<td>Clock Domain Analysis</td>
<td>50</td>
</tr>
<tr>
<td>6.5</td>
<td>Graph Analysis and Feedback</td>
<td>50</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusion</td>
<td>52</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>55</td>
</tr>
</tbody>
</table>
Appendix A  Code Examples .................................................. 57
  A.1  Netlist in Various Formats ........................................... 57
    A.1.1  Verilog Source ...................................................... 57
    A.1.2  EDIF formatted netlist ........................................... 58
    A.1.3  Verilog Formatted Netlist ....................................... 65
  A.2  Creating a Netlist ................................................... 68
  A.3  Traversing A Netlist .................................................. 71
LIST OF TABLES

3.1 Similar Structures in Various Formats ........................................ 19
5.1 Watched Callback API Methods .................................................. 45
LIST OF FIGURES

1.1 SpyDrNet High-Level Structure ........................................... 2
2.1 Example Netlist ............................................................. 9
2.2 Instance and Definition view ............................................. 11
3.1 SpyDrNet Class Objects Visualized .................................... 20
3.2 Intermediate Representation Relationships .............................. 23
3.3 Intermediate Representation Formats .................................. 25
3.4 Consistency in the Intermediate Representation ...................... 26
3.5 Available Getter Functions .............................................. 31
5.1 Callback Positioning ....................................................... 40
5.2 Namespace Organization .................................................. 42
6.1 TMR Voter Options .......................................................... 49
6.2 DWC with Duplicated Comparators ..................................... 50
CHAPTER 1. INTRODUCTION

Advanced digital circuits have become an increasingly important part of our lives. These designs are frequently created by several developers working together. These designs are most commonly written at the register transfer level (RTL) in a language like Verilog, VHDL, or in other high-level languages like C++. These designs are synthesized into a structural netlist, which is in turn synthesized into low-level FPGA implementation files or other description files that can be programmed to a device or sent to be fabricated into a physical chip.

After synthesizing a structural netlist, designers might feel that a structural change to the design would improve it. If the source files are not available the designer may wish to directly modify the netlist. Even if the source files are available it may still be desirable to modify the netlist instead. For example, source files can be complex and should be well tested, so modifying the source may require a potential redesign phase of a project. In other cases, a programmatic change to the netlist that can be applied to multiple netlists without additional overhead might be desired.

1.1 SpyDrNet: Direct Netlist Modifications

Editing a netlist file manually can be a time consuming, error prone and a difficult task to do by hand. SpyDrNet is a tool that makes these edits simpler. It provides users with parsers, composers, and an intermediate representation capable of handling several netlist formats. Applications and utilities have been and can be designed on the intermediate representation. This lets users contribute to and leverage a library of netlist modification utilities without redoing the parser or composer steps. Alternatively, users can create a new parser and composer and leverage existing applications.

SpyDrNet provides flexibility for designers to create algorithms to incorporate into their tool flows through a modular design pattern and open-source nature. SpyDrNet is centered around
a simple intermediate representation designed to hold all available data while prioritizing the most significant structural information. The associated intermediate representation application programming interface (API) abstracts away error prone pointer changes and helps motivate simple idioms for modification. Xilinx EDIF and Verilog tool flows have both been included to provide support for associated devices and illustrate SpyDrNet's usefulness.

SpyDrNet is a large project with several contributors. The main components are illustrated in Figure 1.1. Note that SpyDrNet is composed of parsers, composers, API functions, and applications for intermediate representation modification. Specifics about these components are discussed in future chapters.

![Figure 1.1: Shows the high-level components that make up SpyDrNet](image)

### 1.2 SpyDrNet Features

SpyDrNet incorporates the benefits of open-source software and the Python language to help aid designers to algorithmically change their netlist designs. At the core of SpyDrNet lies its intermediate representation. All other components were built around the intermediate representation. The intermediate representation maintains a generic format to represent any structural netlist. Most of the elements of the intermediate representation are expandable through metadata.
dictionaries and are mutable. These additions simplify creation and modification of portions of netlists. The intermediate representation is described in more detail in Chapter 3.

1.2.1 Open Source Nature of the Tools

SpyDrNet was created as an open source tool. Being open source adds to the appeal of the SpyDrNet representation and framework. Open-source software gives users the ability to modify the tool as needed to fix bugs that they deem a priority, with or without maintainer support. It also allows users to verify the data structure and low-level changes are doing what they expect. Although it is less ideal than well maintained documentation, the source code can provide additional insight into how the tool can be leveraged.

Additionally the open source nature of SpyDrNet promotes the construction of custom tools and applications. Users can leverage SpyDrNet and include it in their tool. Because its source is available, users can easily customize how SpyDrNet is incorporated into their tool. Applications and studies have been done leveraging the SpyDrNet tools as illustrated in Chapter 6.

1.2.2 Leveraging the Python Language

Python is one of the most popular and widely used programming languages for open-source projects. It aims to allow rapid development and deployment. Currently, some sources list it as the second most popular language used in pull requests and pushes on GitHub, a popular open-source project sharing platform [1]. By using Python, SpyDrNet is poised to gain widespread usage and improvement especially as an open-source tool.

SpyDrNet is built in Python to leverage the wide availability of many libraries and tools that are available in Python. At least one user of SpyDrNet leveraged a graph visualization program to visualize netlists in SpyDrNet [2]. Another user leveraged the package NetworkX to better understand strongly connected components in netlists [3]. Various other packages could be leveraged to understand and analyze netlist data structures.

Python’s package manager Pypi, that provides the ability for users to use pip, allows users to install SpyDrNet directly from the command line on any computer or device that supports Python.
for free [4]. This eases installation and usage of SpyDrNet. It is also very convenient to upload packages to pip. These factors make pip an excellent package management system for SpyDrNet.

1.2.3 Additional Features

Built into SpyDrNet’s intermediate representation is an API. This interface gives users a useful handle to the intermediate representation that will keep track of the references behind the scenes. Additional quality-of-life features have been added to the API. Clone functionality is built in to aid in circuit replication and modification, getter functions provide quick traversal through netlist structures, and hierarchical references allow users to quickly retrace their steps toward the root of the hierarchy in a unique way.

Although SpyDrNet is extensible to many netlist formats, SpyDrNet already includes end-to-end support (parsers and composers) for two common netlist formats: Verilog and EDIF. These parsers introduced several interesting challenges. An EDIF parser was created first, proving the efficacy of the intermediate representation. A Verilog format parser (for the structural or netlist subset of the format) was then added. Structurally equivalent netlists were then parsed with their respective parsers and compared. Several format specific constructs arose, including Verilog assignment statements, differences in naming restrictions, and how ports were implemented. This exercise shed light on the decision-making process required when implementing additional languages discussed further in Chapter 4.

SpyDrNet has been used for several applications, including circuit replication to increase reliability, clock domain analysis, graph analysis and netlist visualization. These applications have been implemented by several users of and contributors to the open-source SpyDrNet framework. These applications highlight the utility of SpyDrNet.

1.3 Contributions of this Thesis

This thesis documents the author’s major contributions to SpyDrNet and resulting advances to the state-of-the-art in netlist parsing, manipulation, and composing. Parts of this thesis discuss work from contributors other than the author to illustrate the usefulness of SpyDrNet and to give a more complete view of the framework to the reader. Sections which discuss these features cite
sources to credit the other contributors where possible. As a maintained open-source project it is likely that additional contributors will continue to modify and improve on the features discussed here. Although credit cannot yet be given to future contributors, their efforts are appreciated. The following is a list of the major contributions attributable to this thesis:

- **Intermediate representation** The author of this thesis was involved in the SpyDrNet architecture design and decision making process at the inception of SpyDrNet as to how the intermediate representation should operate in conjunction with Andrew Keller, Michael Bjerregaard, and Dr. Mike Wirthlin. Jordi Ramos Chen later contributed many improvements. As an example, the author primarily contributed the clone functionality, bug fixes, and design decisions. Keller was the primary contributor to the getter functions, hierarchical references and much of the intermediate representation structure. Many shortcuts and quality of life improvements were created by Ramos Chen. See Chapters 3 and 5 and the work published in SciPy 2020 conference [5] for additional details on these contributions. Additional work is currently being pursued by the Configurable Computing Lab at Brigham Young University under the direction of Dr. Mike Wirthlin.

- **The structural Verilog parser and composer (an initial alpha version and a more robust version post release 1.7).** This contribution is discussed in Chapter 4. The author contributed the implementation of these components. This feature was key in illustrating the utility of SpyDrNet. It doubled the format handling capabilities of SpyDrNet and led to improvements in the intermediate representation and EDIF parser and composer that made it more representation independent. Although the Verilog parser follows the same tokenizer/parser structure as the EDIF parser, its implementation was non-trivial due to the variety of differences between the Verilog and EDIF formats.

- **The EDIF format support.** The author was the primary contributor of the EDIF composer. This includes later improvements that generate legal EDIF identifiers from Verilog identifiers and a post ordering of libraries and definitions to ensure proper output formatting. Other fixes have been contributed by others. Details of these improvements are found in Chapter 4. The EDIF parser was implemented by Keller [6]. The author improved upon the initial parser by adding the ability to combine single bit nets, as supported by EDIF, into multi bit cables.
for SpyDrNet’s intermediate representation. Corresponding code was added to individualize cables in the composer.

- **Conversion between Verilog and EDIF** The author converted some EDIF and Verilog files back and forth using the SpyDrNet tools. This provided insight into differences between EDIF and structural Verilog. This knowledge was applied in improving the parsers and composers and is discussed in Chapter 4.

- **Clone** The author contributed the clone functionality to the SpyDrNet tools providing assistance in a variety of algorithms and additional functionality. The clone feature is able to copy any of the objects in SpyDrNet’s intermediate representation including each object’s children. Although shallow and deep copy functions can be found in libraries, the SpyDrNet intermediate representation required a custom implementation to ensure that only the pointers of importance would be copied and updated. Considerations included the following: relationships to parents and siblings are severed, ensuring that each object is independent, if an instance and its definition are both copied, the instance will reference the new copy rather than the existing copy and, cables and pins will update references if their sibling objects are also cloned. The clone functionality and its features are discussed further in Chapter 3.

- **Uniquify and Flatten** The author contributed both a uniquify algorithm and a flatten algorithm (with methodology input from Keller). The uniquify function is one of the most frequently leveraged features of SpyDrNet. It has been used to simplify additional algorithms including triple modular redundancy and duplication with compare [6], [7]. The flatten feature was added to allow users to mimic the functionality of the BYU EDIF Tools [8]. Uniquify and flatten are discussed in more detail in Chapter 6.

- **Callback framework** The author was the primary contributor of the callback framework. This framework is included in the intermediate representation and provides users with a simple way to register listener functions that can be alerted when a change is made to the intermediate representation. The callback framework was leveraged to implement a namespace manager to check for namespace collisions and other illegal identifiers that would cause
composed netlists to be less useful. Further discussion on the callbacks and namespace manager is included in Chapter 5.

In summary, SpyDrNet has become a great solution for modifying netlists. Due in large part to the contributions described herein, SpyDrNet is now a viable product for commercial and academic use. Its rich feature set makes it useful for multiple applications, and the open-source design and continued development by Brigham Young University’s Configurable Computing group under Dr. Mike Wirthlin allow for continued improvements over time.
CHAPTER 2. BACKGROUND

In order to best understand SpyDrNet it is important to understand netlists. This chapter describes some of the principles behind netlist structures including connections, encapsulation, and hierarchy. Some of the popular formats for representing netlists are highlighted and some of the similarities and differences are discussed. Additionally some of the tools that are used to represent and modify netlists are outlined including how they compare to the work in this thesis.

2.1 Netlists

As the name suggests, netlists are primarily composed of lists of nets. A net is a connection between circuit components. These components could be any physical electrical component or an abstraction that represents part of a functional unit such as a port on a user defined component. These abstractions are discussed further in Section 2.1.2. Netlists describe a wide variety of electronic circuit designs encompassing both digital and analog circuits. SpyDrNet was designed to represent and operate on digital circuit designs for FPGAs (although many principles would apply to ASIC designs as well).

2.1.1 Nets and Components

The fundamental elements of a netlist are nets and components. A net is a connection between usually one source and its sinks, although exceptions exist for multiple sources. These connections carry a signal, generally a high or low signal between the source and sinks. Nets can connect various components to each other, or connect pins on the packaging of the FPGA to internal components. If several nets have a related purpose they can be combined into a multi bit bus, called a cable in SpyDrNet.

Components leverage a series of pins to allow nets to connect. Component’s descriptions may include some configuration information about how an implementation tool should treat the
physical component to best implement the desired logic. Since a standard component may be used more than one time, this description can be split into two parts to simplify usage. The first part is used to describe the general structure of the component including its source (input) and sink (output) pins, and possible configuration fields. This is called a cell in EDIF, a module in Verilog, or a Definition in SpyDrNet. The other part describes how the component fits in with its neighboring components, this is often called a cell instance in EDIF, module instance in Verilog, or instance in SpyDrNet.

Pins show up both on the definition and instance of a component. They are in the definition as a part of the overall description and to aid in allowing for hierarchy as outlined in Section 2.1.2. They are included in the instances to connect to nets. Pins with a similar purpose are often grouped into ports to help maintain readability of the design and consolidate information about the port, such as if the pins are inputs or outputs to the definition or instance.

Figure 2.1: A portion of a digital netlist visualized in Xilinx Vivado. Components (yellow boxes) with ports (text in the yellow boxes) and nets (green lines) are illustrated. The net named ddata1[3] has been selected to show more information in a call out box.
Figure 2.1 shows a schematic of a netlist structure. The yellow and blue boxes and triangles represent instances of components. Some examples of instances include “ddata_gen.3.UD1”, “reset_ibuf”, “ddr_data_gen.3.U1_IDDR”. Each of these components has ports. For example, the component called “ddr_data_gen.3.U1_IDDR” has “C”, “CE”, “D”, “R”, “S”, “Q1”, Q2” ports. In this schematic input ports are on the left while output ports are shown on the right. Each port is connected to a green line that represents a net. One such net has been selected named “ddata1[3]”. This net runs from the port “DATAOUT” on “ddata_gen.3.UD1” to “D” on “ddr_data_gen.3.U1_IDDR”. Although they are not shown, each net has a name. The text below each of the instances is the name of the definition that is instanced. Note that that a definition can be used more than one time, for example “IDDR” is the definition used to describe both “ddr_data_gen.3.U2_IDDR” and “ddr_data_gen.3.U1_IDDR” instances, thus giving them the same port names. The blue “clk_ibuf” instance has a plus symbol in the upper left corner of the instance. This instance is an instance of a hierarchical definition described in Section 2.1.2.

2.1.2 Hierarchy in Designs

Some netlist structures can hold many components and nets. To help encapsulate and abstract some of the complexity out of the netlist structure, several nets and components can be combined into a single definition that can then be instanced in the design. These instances, with their connections can be further abstracted by other definitions to an arbitrary depth. The opposite of a hierarchical netlist is a flat netlist. Flat and hierarchical netlists can be structurally and logically equivalent.

Figure 2.2 shows an example of this hierarchical structure. The window on the left of the figure shows the visualization of a netlist schematic including an instance, “sub_inst” among others. This instance is blue with the plus symbol in the top left corner illustrating that the instance’s associated definition, “sub”, encapsulates other instances. The definition “sub”, is shown expanded in the right hand side of the figure. The ports of this instances are shown in both views, connecting to external neighbors on the left, and internal components on the right. The definition “sub” also contains another instance “sub1_inst” that represents multiple internal instances in its definition “sub1”. The definitions “sub” and “sub1”, although similarly named, are not the same. This is important because when creating circuits, as a recursive structure would create a circuit which
Figure 2.2: A component named sub_inst is shown highlighting both an instance (left) and definition (right). Primitive components are shown in yellow and non-primitives are blue. Note that non-primitive components can contain other non-primitive components.

could not be implemented. The definition “sub1” and all of the instances which it contains must not include an instance of “sub”, or “sub1” for the circuit to be created.

Keeping hierarchy in a netlist has several benefits. The netlist can be organized to simplify readability by hiding implementation details and allowing repeated blocks to be reused. When they are written to files, netlists that contain hierarchical information can be much more compact than files representing equivalent flat netlists. Some tools keep hierarchical information in the netlist while other tools remove this information and flatten the design. SpyDrNet keeps hierarchical information, however a flatten algorithm is included as described in Chapter 6.

2.2 Netlist Text Representations

The primary purpose of a netlist is to be used as an interchange format between design tools. In most FPGA tool flows, a netlist file is the product of a synthesis tool that is then input to a implementation or analysis tool. Although they are primarily intended to convey information from one tool to another, readability is an important feature of netlists because users rely on predictable names and structures to interpret and verify tool results. This is one reason why netlist formats
generally retain human readable identifiers. Verilog, EDIF, and BLIF are among the more popular formats to represent netlists.

2.2.1 Verilog

Verilog is a commonly used hardware description and verification language that is described in IEEE Std 1364 [9]. Verilog has a wide variety of features that can be used to design and verify circuits. One of these features is the ability to define and instantiate modules and their connections. This feature is essential to netlist structures.

Because of its wide variety of features, Verilog is most commonly used for hardware description and verification. Designers write Verilog source files, using a synthesizable subset of the description language, that describe the circuit they would like to create. The source files can then be tested with other Verilog files that form a test bench for the circuit. After a designer is satisfied with their design, they can process their Verilog source through a synthesis tool to generate a netlist file that contains a structural view of the design. The output of the synthesis tool can be another subset of Verilog or a different netlist format.

This thesis is primarily interested in the subset of Verilog used to represent netlist structures. This subset leverages Verilog’s module definition and instantiation features along with its ability to define nets. Verilog differs from other formats in its description of ports and pins on module definitions and instances. Ports imply the use of a net and assignment statements can be utilized to effectively alias a net in the design. Some synthesizers use constant values in their netlists for low and high signals as well. More information on Verilog specific constructs and how SpyDrNet handles them can be found in Chapter 4.

Appendix A in Section A.1 has an example Verilog netlist generated from a two bit adder design.

2.2.2 EDIF

EDIF (Electronic Design Interchange Format) is a file format designed to represent netlists and schematics. The latest version of the EDIF standard is version 4.0.0 found in "EDIF Version 4
The format has a wide variety of features, including support for circuit layouts and PCB designs. Some of these features are not needed to describe a netlist as used in this work.

Since it was intended for netlist representations, it has a limited feature set when compared with Verilog. Even as a netlist representation EDIF has a relatively limited name space available for identifying components and nets, restricting identifiers to 256 characters and combining identifiers that differ only on case. EDIF requires that objects be defined before they are referenced simplifying parsing. Some identifiers that would be valid or distinct in Verilog must be modified to remain valid or distinct in EDIF. EDIF also restricts nets to a single bit wide. Tools, including SpyDrNet outlined here, use naming conventions to best represent multi bit nets, or complex identifiers in EDIF. Even with these limitations, EDIF is an effective format to represent netlist structures.

Appendix A in Section A.1 has an example EDIF netlist composed from a two bit adder design.

### 2.2.3 BLIF

The Berkeley Logic Interchange Format (BLIF) is a netlist representation format designed by the University of California Berkeley. BLIF was designed to represent netlists and therefore has fewer features for other applications than EDIF or Verilog [11]. This makes the format simpler.

Some features are included that help specifically with digital netlists. The ability to distinguish between clocking and data signals is included. Gates (primitive logic components) and models (hierarchical containers) are used in a similar way to Verilog modules and EDIF cells with the added ability to distinguish between user defined and primitive components. Unlike EDIF and Verilog, flip flops and latches are described explicitly in the format without relying on naming conventions or existing libraries. BLIF also provides a way to distinguish components as technology mapped when they are tied to a specific target device. These features make BLIF an interesting format because it avoids ambiguity that can be caused when naming convention alone distinguishes some of these components.

Although a BLIF parser and composer are not currently included in SpyDrNet, the format is worth mentioning because it is distinct from both EDIF and Verilog but accomplishes the same purpose. SpyDrNet’s intermediate representation is well equipped to handle the specifics that BLIF
provides in the metadata while the structure can be handled with SpyDrNet’s objects. Ongoing work involves adding a parser for an extension of BLIF called eBLIF to SpyDrNet [22]. For more on the capabilities of SpyDrNet’s intermediate representation see Chapter 3.

2.3 Netlist Modification Software

Several tools can represent a netlist in a data structure. This type of software differs from the formats discussed previously because it generally parses one of the netlist formats above. The parsed netlist is held internally in a netlist data structure, allowing the software to modify the netlist structure, append components, and analyze connections.

2.3.1 Verific

Verific is a proprietary netlist representation framework. It can be incorporated into other tools to help manage netlist manipulation. According to their website, “Verific’s Parser Platforms are in production and development use today at numerous companies worldwide, from EDA start-ups to established Fortune 500 semiconductor vendors.” Some products which leverage the Verific platform include several products by Mentor, Synopsys, Xilinx, and Quartus II from Altera. [12]

Verific is a mature library, as such it is capable of parsing and synthesizing several advanced features in VHDL and SystemVerilog files. To support these features, Verific has several additional components in its netlist representation format. These components include built in operator and primitive types and attributes. The format has its own terminology, specifics can be found in the documentation [13]. Verific interfaces for scripting in several languages, including Perl, Tcl, and Python.

Verific users need a username and password to access the documentation, and a license file to access binaries. It requires the user to request a trial with the additional restriction that “[t]he library files have a built in time-bomb.” [14] Licenses can then be obtained for long term usage.

2.3.2 LiveHD

LiveHD [15] is an open-source tool produced by members of the University of California, Santa Cruz’s Department of Computer Science and Engineering. According to their documenta-
tion, “LiveHD is an infrastructure designed for Live Hardware Development.” Meaning that for small changes, it can yield “the synthesis and simulation results in a few seconds.” [16].

LiveHD’s netlist representation component is called LGraph. LGraph’s primary focus is rapid turn around for design changes to be reflected in the synthesized netlist. To achieve this, the structure employs Nodes, Pins, and Edges in a memory mapped file on disk. In hierarchical designs, nodes may be nested [17]. LiveHD is developed in C++ to help it run quickly. This approach is useful for rapid results but may pose a barrier to rapid plugin development.

2.3.3 BYU EDIF Tools

BYU EDIF Tools was created in the same lab as this work and the SpyDrNet tool [8]. It can read netlists in the EDIF format and apply a number of algorithms to the netlist structure before writing the changes to a new file. Most of the algorithms available relate to reliability applications but other algorithms could be created.

The BYU EDIF tools provide two benefits. First, it provides an API for working with electronic design interchange format (EDIF) netlists. Second, the BYU EDIF Tools includes the Brigham Young University and Los Alamos National Laboratory Triple Modular Redundancy (BL-TMR) Tool. The BL-TMR tool provides a rich set of features for the automated insertion of circuit redundancy for the application of fault-tolerance techniques on digital hardware circuits. These tools have been used extensively in FPGA reliability research [18]–[20].

The BYU EDIF Tools have limitations that motivate the development of SpyDrNet. First, the framework of the BYU EDIF Tools is closely tied to the EDIF netlist format, which makes it challenging to use with alternate netlist formats. Second, the BYU EDIF Tools are primarily intended for use with netlists targeting specific FPGAs. Finally, though not a limitation per se, the BYU EDIF Tools are written in Java and migrating to Python is a motivating factor.

2.3.4 SpyDrNet: Netlist Representation, Manipulation, and Analysis Framework

SpyDrNet is an open source project that has an intermediate representation for any generic netlist structure along with an API allowing users to manipulate the netlist structure before it is composed or written out to disk. SpyDrNet differs from each of the Frameworks outlined above.
Verific and SpyDrNet have similar purposes. Both can hold netlist data structures from multiple formats. They both allow analysis and modification of the netlist structure. Both can interface with Python programs to allow users to modify the netlist structure. Verific is more mature and feature rich than SpyDrNet. SpyDrNet is open source however, which is advantageous for some use cases.

LiveHD and SpyDrNet have much in common. They both have an intermediate representation to represent a digital design. Additionally they are both open source. SpyDrNet and LiveHD have different purposes. LiveHD is intended to help with live development of a hardware design. On the other hand, SpyDrNet is intended to provide users with a framework on which to build algorithmic changes to netlist data structures.

The BYU EDIF Tools and SpyDrNet share similar purposes and scopes. Some differences do exist, however. SpyDrNet is written in Python, which is a scripting language aimed at rapid development. It was also designed from the ground up with multiple formats in mind. The BYU EDIF Tools, having been around for longer, have a wider array of applications built on them although this gap is shrinking.

2.4 Subsequent Chapters

The introduction and this chapter introduced foundational concepts related to the development of SpyDrNet that are essential to the story presented in later chapters. The remaining chapters discuss the SpyDrNet development advances contributed by this thesis. First Chapter 3 describes the SpyDrNet intermediate representation, a data structure that represents netlists and allows them to be modified. Additionally this chapter outlines how the netlist data structure can be manipulated once it is in memory. This chapter demonstrates SpyDrNet’s manipulation versatility by showing that SpyDrNet can generate netlists from the ground up. It is key to understanding SpyDrNet and will help give the reader a basis on which to build the following chapters.

Design considerations were made to ensure SpyDrNet could handle multiple formats; this is discussed in Chapter 4. Chapter 5 discusses how callbacks are used to monitor netlist structure changes and ensure they don’t break other aspects of the data structure. It also discusses the namespace manager that uses these callbacks. Chapter 6 includes several applications of the SpyDrNet framework.
In summary, digital circuit design tools leverage netlists to represent the structural design of a circuit. These netlists can be analyzed and modified to improve circuit designs. Netlists have a general structure that includes instances and definitions allowing them to represent hierarchical circuit designs. Additional connection information is present in the form of nets. Several formats can represent netlist data structures. These formats each have a unique style and approach, but they maintain the same key structural items. The similarity in structure between the formats is the kernel around which SpyDrNet and its intermediate representation are built.
CHAPTER 3. SPYDRNET AND ITS INTERMEDIATE REPRESENTATION

SpyDrNet provides a mutable data structure capable of maintaining and operating on netlists from various formats. Mutability, format independence, and flexibility were the primary motivating design objectives for the data structure referred to as the intermediate representation. This motivation resulted in a data structure that is well suited for representing many common netlist formats and making modifications to them. Both the EDIF format and the Verilog format can currently be parsed and written using SpyDrNet. The first portion of this chapter focuses on the structure of SpyDrNet’s in memory netlist representation format. The remainder discusses in greater depth the API, that defines user access points, associated with the intermediate representation. Code examples are found in Appendix A.

The intermediate representation is the nucleus around which existing modification and analysis passes can easily be applied to designs from any format that can be successfully parsed. This makes SpyDrNet an appealing framework with which to build netlist analysis and manipulation tools because it will work into the future and across formats.

3.1 Intermediate Representation Terminology

While SpyDrNet’s structure will be familiar to users of other netlist representation formats and modification frameworks, some of the terms used to refer to components in SpyDrNet are different than those used in other formats. Table 3.1 relates some of the terms used in SpyDrNet to terms used in other formats. SpyDrNet uses an object oriented design pattern for convenience in keeping track of information and its relationships. In other words, each of the SpyDrNet netlist elements types shown in Table 3.1 (netlist, definition, instance, etc.) is defined through an object-oriented class in Python.

All netlists representations, by necessity, can be iterated from the top instance downward in the hierarchy. SpyDrNet is no different. However, for convenience, SpyDrNet objects have
parent pointers that allow users to iterate through the structure from any point in the hierarchy both upward and downward. In other words, SpyDrNet instance objects have a pointer to the definition in which they reside. This concept was extended to objects beyond the definitions and instances that implement hierarchy. For example, a cable object (discussed later in this section) contains a pointer to the definition in which it resides. These pointers are called parent pointers. Pointers that go in the opposite direction are called child pointers or pointers to child objects.

Figure 3.1 shows the terminology used by SpyDrNet applied to a simple circuit design. The netlist contains all information associated with the circuit design including libraries and a top instance. The library in this design contains three definitions. Each definition is itself unused until it is referenced by an associated instance object. Each instance object indicates which definition object it is associated with using a “*”. Inner pins are labeled in the widget definition but are also present in the AND2 and OR2 definitions. The figure does not label outer pins but they are included in each instance present to connect to wires, and in the case of the top instance, to be constrained to inputs and outputs on the physical device.

### 3.1.1 SpyDrNet’s Netlist Class

Chapter 2 outlined that a netlist is a collection of components and their connections. At a minimum then, a netlist requires instances, their definitions, their inputs and outputs and connections between them. Many of the common netlist file formats contain more information than the minimum required. For example, EDIF files may include a design name, libraries of definitions, and

<table>
<thead>
<tr>
<th>SpyDrNet</th>
<th>Verilog</th>
<th>VHDL</th>
<th>EDIF</th>
<th>BLIF</th>
</tr>
</thead>
<tbody>
<tr>
<td>netlist</td>
<td>.v file</td>
<td>.vhd file</td>
<td>.edf file</td>
<td>.blif file</td>
</tr>
<tr>
<td>definition</td>
<td>module</td>
<td>entity/architecture</td>
<td>cell</td>
<td>model</td>
</tr>
<tr>
<td>instance</td>
<td>module instantiation</td>
<td>component instantiation</td>
<td>instance</td>
<td>model reference/subckt</td>
</tr>
<tr>
<td>port</td>
<td>port</td>
<td>port</td>
<td>port</td>
<td>terminal</td>
</tr>
<tr>
<td>innerpin</td>
<td>port</td>
<td>port</td>
<td>port</td>
<td>terminal</td>
</tr>
<tr>
<td>outerpin</td>
<td>port map</td>
<td>port Map</td>
<td>portref</td>
<td>formal-actual-list</td>
</tr>
<tr>
<td>cable/wire</td>
<td>net(wire/reg/logic)</td>
<td>signal(bit/std_logic)</td>
<td>net</td>
<td>signal</td>
</tr>
<tr>
<td>library</td>
<td>library</td>
<td>library</td>
<td>library</td>
<td>library</td>
</tr>
</tbody>
</table>

Table 3.1: Shows the names used to represent various similar structures in some netlist representation formats. Although these structures are similar in each format, they are not identical and each have their own limitations. [9], [11], [21]
Figure 3.1: This diagram shows a very simple circuit with associated SpyDrNet class names indicated. Outer pins are not labeled but would be the pins on the instances of the and and or gates within the widget definition. The “*” indicates a reference to a definition.

(that may not all be used in the design), the author name, design date, and the target architecture. To best represent this extra information, SpyDrNet implements a netlist class that can have it’s own name, libraries, and metadata associated with physical design. SpyDrNet’s netlist class objects allows users to easily read in and store multiple netlist structures at the same time.
3.1.2 Cables and Wires

SpyDrNet implements a wire class to represent connections between components. Since connections are an essential part of netlist structures, all netlist formats include constructs that can be directly represented as wire objects. Objects of the wire class carry a single bit signal from one source to its sinks.

To represent signals composed of multiple related bits, SpyDrNet also implements a cable class. Cables are a collection of wire elements. Some formats, but not all, leverage this construct to simplify connections and highlight relationships between wire elements. SpyDrNet includes cable objects for the same reasons. SpyDrNet’s parsers can combine wire elements present in formats without cables into a cable as discussed in Chapter 4.

3.1.3 Definitions and Instances

As described in Chapter 2, EDIF and Verilog both use the concept of definitions (called cells or modules) and instances, allowing designers to represent hierarchical structures and reuse information in multiple contexts. SpyDrNet implements both a definition and instance class, following this pattern. Understanding the relationship between the definition and instance classes in SpyDrNet is crucial to understanding ports and pins described later in this section. It is also the foundation for hierarchical representations.

The definition class holds the list of ports (ports are explained later in this section) on the component and the information about how a component is connected to internal or children components. It also outlines how ports are set up for external connections although those connections are not described in the definition. Metadata and a name field are available in the definition class.

Instance class objects hold information about how a component is connected to its neighboring external components. These objects make up the list of children of the definition. The ports and pins on instance class objects are related to ports and pins in a definition. In other words, definitions describe the contents of a section of a netlist, and instances describe how sections of a netlist connect to cables, wires and thus to other sections of the netlist. There can be zero or more instances for each definition, but each instance must have a single corresponding definition (Spy-
DrNet allows the creation of instances with no definition but these will not be useful in a netlist structure.

### 3.1.4 Ports and Pins

SpyDrNet implements a pin class that holds the connection point between components and their connected wires and cables. The pin class is extended by two additional classes called inner pin and outer pin that hold the connections to definitions and instances, respectively. Because there can be a many-to-one relationship between instances and definitions, and each pin holds connection information, it is important to have unique pin objects in each of the instance objects in a design. Outer pins are used for this purpose and are contained in instance objects.

Port class objects hold a collection of ordered inner pin objects. Various netlist formats refer to individual pins in a port through an index and through the port name on the instance. To support connecting wires to pins in instance objects with outer pins (but not ports), inner pins can be used in conjunction with a built-in dictionary lookup in SpyDrNet’s instance class to find the associated outer pin on the given instance object. This allows users to index a port’s pins, pass the result to an instance class and obtain the associated outer pin.

### 3.2 Object Relationships

Each of SpyDrNet’s object oriented elements maintains pointers to other element types. Figure 3.2 shows all of the relationships present in SpyDrNet’s intermediate representation. Most of these relationships are bidirectional to aid multi-directional netlist traversal.

Parent elements are on the left of their child elements, and child elements are to the right of parent elements. Elements that are horizontally even with other elements show other relationships in the structure. For example, wires and pins maintain references to one another to represent connections present in the netlist structure. The dotted line from the instance element to the definition element represents the reference to the definition which describes the contents of the instance element. The solid arrow between instance and definition shows a connection to the hierarchical parent definition of the instance (the component in which the instance resides).
3.3 Metadata for a Generic Representation

SpyDrNet's intermediate representation aims to hold all data available in the input file so that it can represent any source format and compose an output file that closely tracks the input file. This data includes anything that is not included in the structure of SpyDrNet objects. Some examples are EDIF rename information, inline constraints in a Verilog file or properties on instances or definitions. This data is referred to as metadata.

Netlist, library, definition, instance, cable, and port objects all inherit from the “first class element” class in SpyDrNet. This class provides a name and metadata fields. The metadata is implemented as a Python dictionary, theoretically allowing any Python object to be used as the key and value for the metadata. However, in practice only strings and dictionaries of strings have been tested. Although the implementation is identical for each of these elements, the use cases vary between them. For example, author information may be stored in the netlist metadata, while constraint information could be stored by the cable class metadata.

A naming convention was chosen to prefix dictionary keys with the format name and a period, for example “EDIF.” or “VERILOG.” This naming convention ensures that language specific formats can be handled accordingly if a modification pass needs to handle them, or for composition. This convention also provides users with a predictable way to avoid collisions with parser generated metadata. For example, the following is an inline Verilog constraint that is applied to a definition.
The corresponding metadata entry would be under the key "VERILOG.InlineConstraints" whose value would be a dictionary with a key value pair: "ORIG_REF_NAME": "add1". A dictionary that maps to another dictionary is used to ensure the format specific naming convention is followed without prefixing the constraint key directly, which would slow down look-ups. Parameters (used in parameterized instances and definitions) are maintained in a similar way.

The language specific naming convention helps avoid collisions if elements from multiple netlist formats are used, or if the user’s algorithmic pass stores information in the metadata. Additionally this convention can help when writing an output netlist; composers converting the intermediate representation to an output format can easily find relevant information to the desired output format. Additional information on parsing and composing is found in Chapter 4.

3.4 Independence of the Intermediate Representation

SpyDrNet keeps the intermediate representation separate from the parser, composer, and other portions of the tool, which promotes a tool flow built on SpyDrNet that will work with more than one format. Applications such as a parser, composer, and manipulation tools access the intermediate representation through a well-defined API as described in further detail later in this chapter. This simplifies expansion and maintenance of the SpyDrNet ecosystem because the intermediate representation does not need to be updated to included new formats or tools, and modifications to the intermediate representation do not force updates to parsers, composers, or tools.

An independent intermediate representation opens the door to conversions between formats. This flexibility adds some complexity to parsers and composers which must navigate the nuances of both the source format and the intermediate representation. The intermediate representation’s metadata and focused scope help minimize this issue. Considerations related to parsing and composing multiple formats are discussed in further detail in Chapter 4.
3.5 Mutability

As stated at the beginning of this chapter, mutability was one of the design motivations that drove the development of SpyDrNet. Every one of the elements present in SpyDrNet is mutable. Mutability allows Python objects representing intermediate representation elements to be changed in place without the need to create a new object.

The primary complexity of mutability is when multiple dependent pointers exist in various locations in the data structure. One such example is the relationship between a child instance and the parent definition that instantiates it. When the parent pointer is changed in the child instance, the corresponding child pointer needs to be removed from the prior parent and a child pointer needs to be added to the new parent. This example is shown in Figure 3.4. Having one of the pointers change, but not the other will cause the data structure to become inconsistent as illustrated on the figure’s right hand side. Therefore, a well-defined user interface—the API—is required to avoid this and similar issues.
Figure 3.4: Shows a consistent and inconsistent set of pointers in a parent child relationship. The X is on the pointer that changed resulting in inconsistency.

Although the API provides a safe way to maintain consistency, this may be voided if a user edits a member or method in an unintended way. Python does not provide a private option for member variables, but users can avoid the issue by following Python’s convention of using an underscore to begin the name of a private object. For example, "name" would be a public member variable name whereas "_name" would be a private member variable name. Public members are carefully managed, ensuring that a change to any member will correctly update the rest of the structure. This convention in combination with well maintained public members helps make SpyDrNet’s mutable objects possible and easy to use.

3.6 Netlist Connectivity for Analysis

As mentioned previously, SpyDrNet’s intermediate representation represents an interconnected set of elements. These elements were connected to promote easy traversal and analysis of the netlist structure. Traversal and Analysis could be restated as simply following and observing connections between components in the netlist. This is essential in a variety of applications. Spy-
DrNet’s API was designed to aid users in netlist traversal and analysis. This section outlines these considerations.

Objects maintain bidirectional pointers (as illustrated in Figures 3.2 and 3.4. Maintaining both pointers adds some complexity to the overall structure and API implementation. This inconvenience is overcome by the advantage the additional pointers provide to users for netlist traversal; Users can directly traverse to several connected components (all components can be iterated to through the getter functions, described later in this chapter) in both directions. This allows users to not only iterate through all the children elements of a parent, but users can iterate from one net to a connected component and its other connected nets or iterate from a child to a parent.

SpyDrNet’s netlist traversal capabilities allows netlists to be traversed from the bottom to top and top to bottom without unnecessary loops to lookup connected elements. Iterating in both directions is useful in a wide variety of applications. For example, if a user wants to find ports that are connected to a given port all of the pointers necessary are included in the elements: the port has pointers to its pins, the pins point to the wires that connect to them which in turn point to all pins they are connected to, connected pins then point to the port or instance to which they belong.

The decision was made to use ordered lists to contain the children of elements. This helps to maintain order when order is present in a given format such as EDIF. It also gives users a stable data structure which can be indexed into. Ordered wires in cables, and pins in ports naturally followed from EDIF and Verilog netlist formats; keeping ordered sets throughout other one-to-many parent-child relationships, such as definitions in a library, provides a consistent experience.

Netlist traversal, made simpler through SpyDrNet’s bidirectional pointers, is the main feature the API uses for analysis and display of netlist structures. For example, The API can be leveraged to output a hierarchical view of a netlist, a set of components that could be affected by a change in the value of a particular pin, and a complete netlist structure [2], [22]. These API functions help users more easily modify a design and enable composing a netlist as outlined in Chapter 4.

Appendix A has example Python code of several netlist traversal operations made possible through SpyDrNet in Section A.3.
3.7 API

SpyDrNet’s intermediate representation is accessed and modified through a pre-defined API that helps ensure changes do not cause inconsistencies in the data structure. The API is a well-defined set of accessors and modifiers that are available in conjunction with each of SpyDrNet’s intermediate representation objects.

The API functions are built into the intermediate representation as object-oriented methods. These functions are intended to be static and provide a stable base on which tool flows can rely. The intent is that all necessary functionality is present in the API. SpyDrNet’s intermediate representation API is sufficiently complete to create a full netlist using the available API from the ground up. In fact the SpyDrNet API has been used to create a simple example netlist from scratch [2], [22] as shown in Appendix A Section A.2. Also, the included parsers and composers exclusively use API functions.

3.7.1 Netlist Creation and Modification

The API has a top-down approach to netlist creation. The top most object is a netlist object with no parent. To create a library object, users can call `netlist.create_library()`. Similarly, `library.create_definition()`, `definition.create_cable()`, `definition.create_port()`, and `definition.create_child()` create definitions, cables, ports, and instances respectively. The child element APIs do not feature the ability to add them to a parent.

This design decision was made to help keep the API predictable and easy to use. This reduces the number of functions available, simplifying maintenance and scripting. The top-down approach benefits netlist creation in the parsers for EDIF and Verilog (and likely other formats). When generating a netlist, instead of having to maintain a pointer to the newly created parent item to add to the children when they are encountered, the idioms used to create items keep the parent present.

At the same time, users are also able to create from the bottom-up. It would require that the user first create the child, then create the parent, then use the API methods within the parent to add the parent-child and child-parent pointers. This is not the expected approach, but the functionality is present to allow this.
Code illustrating how to create a netlist is included in Appendix A under Section A.2.

3.7.2 Clone

SpyDrNet’s intermediate representation elements have a built-in clone function as part of the API. Currently all the components in a netlist can be cloned from pins and wires to whole netlist objects. Several factors were taken into account when writing the clone functionality including cloned object connectivity, Python’s built-in deep copy function, coding complexity, and resultant data structure consistency.

Each element in the SpyDrNet intermediate representation required a custom deep copy operator so that the resultant objects would be properly connected. An attempt was made to ensure each object was logically connected as they were cloned. At each level, lower-level cloned objects maintain their connections to cloned sub-elements so that the structure of the cloned object is identical to the original. The cloned object does not belong to any other parent object in the netlist and is not connected to any neighboring components. This ensures that the cloned object is not dependent on the netlist and could be added to any other netlist. There are of course some exceptions to these rules which seemed judicious. One such example is that when cloning an instance, that instance will maintain its original corresponding definition, unless the corresponding definition is also being cloned as in the case of cloning a whole library or netlist (in which case the new cloned definition will be used).

This functionality was accomplished through direct reference modifications, bypassing the creation and modification API. Two functions are included in each element: a private clone function that leaves the objects in an inconsistent state that contains pointers which must be either removed or updated, and a public wrapper that calls the private function and cleans up the result. Parent elements call the private function directly and use the extra pointers in conjunction with a dictionary relating original elements to their cloned versions to ensure inner objects are properly connected and not cloned more than once. This approach simplified the clone functionality’s implementation, promoted code reuse in the clone implementation and helped minimize the number of dictionaries used.

The clone algorithm is especially useful while implementing some of the higher-level algorithms such as TMR and DWC as explained in Chapter 6 that are used for reliability research.
In these algorithms, cloning is essential, and having it built into the SpyDrNet API helps simplify their implementation.

The following example code will clone an element and then add that element back into the netlist which it originally belonged to. Comments are included to illuminate what each step accomplishes. [5]

```
1 import spydrnet as sdn
2 netlist = sdn.load_example_netlist_by_name('hierarchical_luts')
3
4 # index can be found by printing children's names
5
6 # get a handle to the instance sub
7 sub = netlist.top_instance.reference.children[2]
8
9 # clone sub
10 sub_clone = sub.clone()
11
12 # rename needed to be added back into the netlist
13 # this prevents naming conflicts
14 sub_clone.name = "sub_clone"
15
16 # add the cloned and renamed instance into the netlist
17 netlist.top_instance.reference.add_child(sub_clone)
```

### 3.7.3 Getter functions

SpyDrNet includes getter functions which are helpful in the analysis and transformation of netlists. These functions were created to help a user more quickly traverse the netlist. They provide the user with quick access to related components of any element type, even in cases where a direct pointer is not available. For example, it is possible to get a list of all cables present in a Library, or find the Definition in which the port that contains a given pin resides. Figure 3.5 illustrates the ability to get the related elements of any type. Like clone, an attempt was made to ensure logical and consistent rules for the getter functions were used.
Figure 3.5: Each line represents a getter function, showing what related components can be obtained from a starting component. Note that all components can be obtained from a given component.

There are some places in which multiple objects should be returned, for example, getting all pins within a port. In these cases, multiple items will be returned in the generator. In cases in which there are two classes of relationships upon which to return objects, the user may specify whether they would like to get the more inward related or outward related objects. For example, a port may have outer pins on instances or inner pins within the port in the definition. Both of these pins can be obtained separately by passing an inner or outer flag to the getter function. Keller was the primary author of the getter functions within SpyDrNet [6].
3.7.4 Hierarchical References

SpyDrNet includes the ability to create a hierarchical reference graph of all instances, ports, cables, and other objects which may be instantiated. The goal behind hierarchical references is to create a graph on which other tools, such as NetworkX can more easily execute graph analysis algorithms. Each hierarchical reference will be unique, even if the underlying component is not unique (because it is used multiple times in a hierarchy). These components are also very light weight to minimize memory impact since it is advantageous to ensure larger netlists can be represented properly. Keller was the primary author of the hierarchical references within SpyDrNet [6].

The code below shows how one can get and print hierarchical references. The hierarchical references can represent any spydrnet object. [5]

```python
1 top = netlist.top_instance
2 child_instances = top.reference.children
3
4 for h in sdn.get_hinstances(child_instances):
5     print(h, type(h.item).__name__)
```

In summary, SpyDrNet provides a data structure capable of representing netlists from various source formats. Several features were built to enhance the basic API present in SpyDrNet. These features provide functionality to users to help them get the most out of SpyDrNet. These features are leveraged by applications to simplify algorithms and implement frequently used idioms within the core of SpyDrNet.
CHAPTER 4.  PARSING AND COMPOSING

SpyDrNet comes with two parsers and corresponding composers to read and write netlists in the EDIF and Verilog formats (discussed in Chapter 2). One of the motivating goals included the ability to convert a netlist between these two formats, to be able to apply transformation and analysis passes to netlists from both formats in the intermediate representation, and to produce a result that approximates the original netlist file as closely as possible. Several hurdles were overcome to achieve these goals. This chapter describes how the SpyDrNet parsers and composers read from and write to a format that differs from SpyDrNet intermediate representation.

4.1 Parsing and Composing the EDIF Format

Since SpyDrNet aimed to build on the functionality of the BYU EDIF Tools, the EDIF format was originally targeted while maintaining the ability to represent a generic netlist format. The EDIF 2.0.0 specification has three features that differ from other formats. First, identifiers can be accompanied by a rename construct to provide more information than an EDIF identifier can contain. Second, EDIF does not represent multi bit cables directly, instead nets are a single bit wide. Naming conventions are often used to indicate relationships between single bit nets. Third, EDIF formatted files must define cells before they can be instanced. [6], [10], [23].

4.1.1 EDIF Rename Construct

EDIF has a limited name space compared to Verilog and VHDL. EDIF identifiers cannot be longer than 256 characters and they must be case insensitive (although this convention is broken by some tools). They must be composed only of letters, underscores, and numbers, with a letter as the first character unless the name is prefixed by an ampersand, in which case an underscore can be the next character. EDIF files can leverage a rename construct to maintain a more complex name alongside the EDIF name.
The following example is a modified excerpt from b13.edf available as a support file in the Github repository [22]. The net has a rename construct that uses \texttt{S2\_1\_} as the valid EDIF identifier and \texttt{S2[1]} as the complex name.

\begin{verbatim}
(net (rename S2\_1\_ "S2[1]") (joined
  (portref I0 (instanceref add_mpx2_i_1))
  (portref I0 (instanceref mpx_i_1))
  (portref I1 (instanceref rdy_i_1))
  (portref I1 (instanceref shot_i_1)))
)
\end{verbatim}

Since SpyDrNet’s intermediate representation only has a single name associated with each object, the EDIF identifier, \texttt{S2\_1\_} in this case, is maintained in the metadata. This ensures the intermediate representation maintains the rename information for composing an EDIF file, while also allowing all tool passes and other language composers to safely ignore the EDIF valid identifier. Since the construct is only present in the EDIF format there is no additional incentive to add it to the API.

When writing a netlist that may have originated from another format, it is unlikely that the EDIF identifier will be present in the metadata. The EDIF composer is expected to generate a valid EDIF identifier and associated rename construct when they are not present. This is accomplished by truncating a name that is too long, replacing invalid characters and ensuring the generated name is unique by adding a unique identifier if needed. The process is repeated until a unique, valid, EDIF identifier is generated.

\subsection{Single Bit Nets}

The EDIF format provides users with single bit wide nets. This is a limitation compared to the SpyDrNet intermediate representation which provides multi bit cables. Naming conventions are often used in tools to represent multi bit nets in the EDIF format. SpyDrNet combines nets with names ending in bracket enclosed numbers, \texttt{some\_net[0]} and \texttt{some\_net[1]} for example, to make a cable. When only EDIF valid identifiers are present, it will combine nets ending in numbers surrounded by underscores, \texttt{some\_net\_0\_} and \texttt{some\_net\_1\_} for example, When writing an EDIF
file, multi bit cables are written following the same naming convention that would combine a cable when parsed.

### 4.1.3 Ordered Cell Definitions

EDIF cells can have instances of other cells within them. The cells that represent the contents of these instances must be defined before the instances can be used. In other words, cells must be defined before they can be instanced and cells wishing to leverage other cells must be defined after the cell they wish to use. This makes parsing EDIF files more straightforward. This dependency order is kept in SpyDrNet when an EDIF file is read since SpyDrNet’s internal lists are ordered.

When composing a netlist in the EDIF format, this order needs to be maintained. SpyDrNet’s composer implements a post order function that ensures that cells with dependencies are written in the proper order. This allows netlist files originally represented in Verilog, for example, to be composed out in the proper order in an EDIF file. Other source formats being composed in EDIF would also benefit from this feature.

### 4.2 Parsing and Composing the Verilog Format

SpyDrNet aimed to provide support for Verilog netlists generated by Xilinx Vivado. These netlists leverage a subset of Verilog’s features to represent the structural details contained in a netlist datastructure. Appendix A.1 contains an example Verilog netlist file using this subset and the source Verilog RTL description. This subset includes module declarations and instantiations (including ports and parameter mapping), assignment statements, and wire declarations. This section discusses some of these features of interest.

#### 4.2.1 The Assignment Statement

Netlists described in Verilog occasionally leverage the assignment statement, for example, `assign signal_1 = signal_2`. These statements can be thought of as a connection between two distinct cables or an aliasing from one name to the other. Since SpyDrNet does not allow direct connections between cables, these statements are handled by linking the two cables through
a new instance of a simple pass through (assigning) definition generated by the parser. This allows
the original assignment statement to be recovered from the intermediate representation because
both names are maintained. It is also logically equivalent to the original statement and allows both
cables to be used in later logic. When composing an assignment statement into a different format,
the generated assignment instance will be present.

4.2.2 Module Port Declarations

Verilog ports declarations are present at the beginning of modules and may be fully declared
in a port list, or named in the port list and described in a module body. These declarations take
the form of `input [1:0] B;`. The direction keywords `input`, `output` or, `inout` indicate the
direction of the signal into, out of, or both in and out of the module being declared. The numbers
in brackets indicate how many signals can be found in the bus, two in the given example. This is
then followed by the name.

Port identifiers in Verilog can be used in the same places that signal identifiers can be used,
such as port mappings in module instances and assignment statements. This differs from the EDIF
and the SpyDrNet implementation of ports which must be connected to nets or cables respectively.
SpyDrNet is able to maintain this connection information by creating and connecting cables with
matching names to both the ports and their connected components. This ensures that SpyDrNet’s
netlist representation is logically consistent with the source Verilog while maintaining the original
names. This is possible because SpyDrNet has separate namespaces for ports and cables within a
definition. This handling is also format agnostic and does not affect the structure of the represented
netlist even when composing to an alternative format.

4.2.3 Port Concatenations

Some Verilog files generated by Vivado provide different names by which the ports are
referred on the interior and exterior of the module, generally in the form of a collection of names
on the interior representing the individual signals that make up a single port on the exterior. The
following Verilog example has both regular ports, `add_mpx2`, and `clock` and a port set up as a
concatenation of internal signals, `canale`.

36
module b13
  (add_mpx2,
   clock,
   .canale({canale_3, canale_2, canale_1, canale_0})
  );
output add_mpx2;
input clock;
output canale_0;
output canale_1;
output canale_2;
output canale_3;

This construct can be represented in SpyDrNet’s intermediate representation. When parsing the file, the first identifier, canale is used as a port name. The remaining identifiers closed within the braces, canale_3, canale_2, canale_1, canale_0, are used as cable names on the interior of the definition instead of the implied cable name that would be present otherwise, as discussed in Section 4.2.2. The directions are declared next to the cable names and are assigned to the connected port. If multiple directions are specified, a bidirectional, inout, port is assumed. When composing or writing out a Verilog file with this construct, the cables can be checked to see if they are connected to a port and if the port’s name and width match that of the cable. When they do not match the concatenation construct is generated.

Using the metadata and assignment statement logic were both considered prior to implementing the solution outlined here. Using the metadata compromised the generality of the resulting intermediate representation and reusing the assignment statements complicated recovering the original syntax. Handling this by modifying the cables that the port implies was the simplest solution allowing both reproducing the original syntax and remaining general.

### 4.2.4 Parsing Instances Before Definitions

Verilog does not enforce ordering restrictions on module instances and definitions. The flexibility introduces complexity in the parsing step. When an instance of a definition that has yet to be parsed is encountered, the parser infers details that are not yet defined. For example, a module may be instantiated with wires mapped to one or more of its ports as shown in the code example.
SpyDrNet infers that the corresponding definition exists with the given ports of the width of the wires. In this case, a definition for OBUF is created and put into a map which can be searched by name so that future instances can be correlated. The definition will have my_in and my_out added as ports, both with a width of one (in a more general case the width of the cables that are being mapped). When the definition is encountered, if it ever is, the existing definition will be updated to reflect the new information. If the definition is never encountered the inferred definition will be maintained as a leaf cell.

In summary, SpyDrNet is capable of parsing and composing both Verilog and EDIF formatted netlists while maintaining a general intermediate representation. Several format specific constructs were considered and an approach was taken for each construct that ensured a generic, logically equivalent netlist was represented within SpyDrNet. This approach requires careful parsing and composing but it allows users to leverage either parser or composer with out other changes in their workflow. Additionally it makes a conversion between formats seamless for the user.
CHAPTER 5. CALLBACKS AND THE NAMESPACE MANAGER

A callback framework was implemented in SpyDrNet to support real time analysis of netlist modifications. Callbacks can assist with applications that make incremental changes to the netlist followed with an analysis of the netlist to determine what more needs to change. Alternatively, users may wish to be warned of violations of design rules such as maintaining unique names. Callbacks provide a handle to perform these checks incrementally. [5]

The primary use case for the callback framework is to allow callback functions to check for valid netlist conditions and provide warnings if an invalidating change is made. While this is the only use case implemented to date, other applications could be implemented on this functionality as well. The namespace manager included in SpyDrNet provides this checking on namespaces within netlist structures.

5.1 Callback Framework

SpyDrNet’s callback framework executes the callbacks before any change is made to the netlist, and callback order is consistent and user definable. These features make the callback framework an excellent option for users who wish to warn the user of possible design rule violations caused by their changes to the netlist. Classes that leverage the callback framework are referred to as plugins.

5.1.1 Creating a Plugin

Implementing a callback listener plugin involves creating a class that inherits and overrides from the existing CallbackListener class. Each of the desired methods will need to be overwritten in the inheriting class. This class has a stub for every function that can be used. If these stubs are not overwritten but still registered, they will raise an exception indicating that they were not properly overwritten.
As shown in Figure 5.1, callbacks are made after some of the built-in sanity checks (checks to make sure the API can accept the object type and that it is present/absent for remove/add commands) but before modifications are made to the data structure. This allows users of the callback framework to prevent unwanted changes from taking place.

If users create more than one plugin, the callbacks will be made in the order in which the plugins are registered. A future user may, for example, create a plugin similar to the namespace manager that alerts them if more than one instance of any particular definition is created (the netlist loses its uniqueness). If the user registers this callback after the namespace manager is registered, the callbacks to their plugin are ensured to happen after the namespace manager callbacks are called. If the opposite order is desired, the user plugin could be registered before the namespace manager.

### 5.1.2 Interface with the API

The callback framework is called from the API directly. Users can register and watch only the functions they desire. This ability ensures that adding simple watchers does not slow down unwatched functions. Users can also register and deregister plugins on demand without affecting other plugins. If a particular plugin is too slow for a desired operation, it can easily be disabled before the desired operation and enabled after it completes.

### 5.1.3 Watched features

Table 5.1 lists the API function calls that can be watched using the callback strategy. The feature column lists the function/behavior that is watched for. The "Watched SpyDrNet Classes"
column highlights the SpyDrNet class on which the feature is watched. The parameters column lists the objects that are passed back. These functions in the API were chosen for the callback framework because they are the basic functions that allow the user to completely create a new netlist. Many of the other functions in the API rely on these functions, and therefore will have callbacks through these functions. The clone function is an exception to this as a need for clone callbacks has not yet arisen because cloned objects are frequently added back into a netlist with one of the covered functions.

5.2 Benefits of Leveraging the Callback Framework for Namespace Managers

A namespace manager plugin is included with SpyDrNet to warn the user when an invalid identifier is added to the netlist. Through the callback framework, it can watch changes to the netlist, including addition and removal of elements, as well as changes in naming and structure of the netlist. This provides users with exceptions that alert them if an element with an invalid name is added to the netlist data structure. This lets them catch the exception and avoid making an invalid netlist. [5]

Because the callbacks are made before the netlist is modified, changes can be stopped before an invalid name is added with a raised exception which will exit the program unless the user has handled it. The real-time nature of the callbacks allows the namespace manager to alert the user immediately when an invalid operation is made. This eases debugging.

5.3 Namespaces

Namespaces are hierarchical levels or contexts in which a name provides a text reference to another element in the netlist, usually a one-to-one relationship with elements. In other words, namespaces imply a set of unique non-overlapping names in a single scope. There are several individual namespaces within most netlist structures. In the cases of EDIF and Verilog, a global namespace exists for libraries. Each library has its own namespace for Modules or Cells. Modules or cells in turn have their own namespaces for nets/wires ports, and instances.
5.3.1 Organization

The namespace managers provide separate namespaces based on element types. Both the default and EDIF namespaces allow for one namespace for Libraries indexed by the Netlist, one namespace for Definitions indexed by the library and three namespaces for Ports, Cables, and Instances indexed by the Definition. These namespaces are illustrated in Figure 5.2. If a new source netlist format is parsed into SpyDrNet with different namespaces, namespace scopes could be modified by creating a custom namespace manager for that format.

![Namespace Diagram](image)

Figure 5.2: This figure illustrates how namespaces are organized in SpyDrNet’s namespace manager. Grey boxes represent individual namespaces within the scope of the white boxes.

The namespace manager checks each of the namespaces when an object is added, removed, or renamed to ensure that the new name does not clash with an old name. Since the name member of SpyDrNet objects is stored in the metadata dictionary, name comparisons are made against the "NAME" entry in the dictionary. This provides a template on which additional keys could be watched. The EDIF namespace manager rule set leverages this feature to watch for collisions on the "EDIF.identifier" key.

The default manager accepts any name and has no name formatting restrictions, but it has functions in place to allow a non-default namespace manager to override the default when it is extended. The EDIF manager overrides the default format checks to enforce EDIF naming con-
ventions. Verilog currently uses the default namespace manager. This provides collision checking but naming rules are not enforced.

5.3.2 Manager

The namespace manager leverages the callback framework, and therefore is a plugin. It is loaded when the SpyDrNet Python library is loaded. It will constantly be checking names and rejecting invalid or duplicate names. The active namespace can be set in code to allow for parsing in a different language or in other cases where a valid name in the namespace may differ. If desired the namespace manager’s callbacks can be unregistered to disable it.

The namespace manager allows new namespace rule sets to be created for new formats and naming conventions. These rule sets can be changed to support checking for valid names in a variety of source netlist formats. The format can be selected prior to setting parsing as shown in the example code in Section 5.3.3.

5.3.3 Using the namespace managers

Currently only "DEFAULT" and "EDIF" are available namespaces. Users can select one of these namespaces before parsing a new file or otherwise creating a new netlist. The following code will change the namespace of the namespace manager, parse a file, apply a modification to the netlist, and write the resulting netlist back out. Throughout each of these functions if an invalid name is encountered (for example in the source file) the namespace manager can alert the user.

```python
from spydrnet.plugins import namespace_manager

# save the default just to be safe. (optional)
ns_default = namespace_manager.default

# can be set to "EDIF" or "DEFAULT"
namespace_manager.default = "EDIF"

# new namespace rules will apply
netlist = sdn.parse("myfile.edf")
sdn.uniquify(netlist)
sdn.compose(netlist, "out.edf")
```
# set the default back (optional)

namespace_manager.default = ns_default

The callback framework and namespace managers are present to help make utilizing SpyDrNet more robust. Currently they are not intended to be disabled. The callback framework is included in the API, adding a dependency to the intermediate representation but affording users the ability to create modular plugins that watch netlist modifications in real time. The namespace manager is also included and registered by default in SpyDrNet. It helps ensure that names do not conflict, and that names follow the rules that are applicable to their namespace. The overhead created by implementing the callbacks and namespace managers has not been found to be prohibitive and the additional functionality and sanity checks are desirable.
Table 5.1: Highlights which methods are watched by the callback framework, Key: N - Netlist, L - Library, D - Definition, P - Port, C - Cable, I - Instance, W - Wire, p - Pin ** currently the pins do not have any callbacks

<table>
<thead>
<tr>
<th>Watched API Method</th>
<th>Watched Classes</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation</td>
<td>NLDPCIW-</td>
<td>Created object</td>
</tr>
<tr>
<td>Add/Remove Wire</td>
<td>—C—</td>
<td>cable, wire</td>
</tr>
<tr>
<td>Add/Remove Port</td>
<td>—D—</td>
<td>definition, port</td>
</tr>
<tr>
<td>Add/Remove Child</td>
<td>—D—</td>
<td>definition, child</td>
</tr>
<tr>
<td>Add/Remove Cable</td>
<td>—D—</td>
<td>definition, cable</td>
</tr>
<tr>
<td>Modify Reference</td>
<td>—I—</td>
<td>instance, reference</td>
</tr>
<tr>
<td>Add/Remove Definition</td>
<td>-L——</td>
<td>netlist, instance</td>
</tr>
<tr>
<td>Add/Remove Netlist</td>
<td>N——</td>
<td>netlist, library</td>
</tr>
<tr>
<td>Add/Remove Pin</td>
<td>—P—</td>
<td>port, pin</td>
</tr>
<tr>
<td>Connect/Disconnect Pin</td>
<td>—W—</td>
<td>wire, pin</td>
</tr>
<tr>
<td>Set/Delete/Pop metadata</td>
<td>NLDPCIW-</td>
<td>element (object), key</td>
</tr>
</tbody>
</table>
SpyDrNet may be used for a wide variety of applications. SpyDrNet grew out of a lab that is focused primarily on improving circuit reliability and security. An application that has had strong influence over its development is that of enhancing circuit reliability in harsh radiation environments through circuit replication [6], [19].

SpyDrNet was created to help automate this process and allow researchers to spend more time studying the resulting improved circuitry and less time modifying the circuit itself. One common adjustment to a netlist for reliability purposes, is a replication of various components. Below are some details on using SpyDrNet for higher level transformation and analysis techniques applicable to reliability or other applications [5].

6.1 Flatten

As discussed in Chapter 2 netlist designs can contain hierarchical information. This lets netlists encapsulate structures within an individual cell that can be referenced or instantiated elsewhere in the design. A flat design is one which does not contain any hierarchical levels within it. In other words it is a design that only contains leaf components. The process of flattening takes a design with hierarchical information and removes the hierarchical components, replacing them with the components which they represent.

SpyDrNet can flatten hierarchical designs. One method to remove hierarchy from a design is to move all the sub-components to the top level of the netlist. This process is repeated until each sub-component at the top level is a leaf instance. Leaf instances are instances where no more structural information is included as a sub-component of that instance.

Flattening was added to SpyDrNet because there are some algorithms which can be applied more simply on a flat design. These algorithms include Graph analysis, and other algorithms where
the connections between low level components are of interest. An example is included below of how one might flatten a netlist in SpyDrNet. [5]

```python
import spydrnet as sdn
from sdn.flatten import flatten

netlist = sdn.load_example_netlist_by_name('fourBitCounter')

# flattens in place. netlist will now be flat.
flatten(netlist)
```

## 6.2 Uniquify

Uniquify ensures that each non-primitive instance is unique, meaning that each definition corresponds to a single instance and each instance corresponds to a single definition. Non-unique definitions and instances may exist in most netlist formats.

The uniquify algorithm is especially useful when modifications are desired on a specific part of the netlist but not to all instances of the modified component. For example, in a four-bit adder, the most significant bit may not need a carry out. The single-bit adder in that position could be simplified. However, if we make modifications to the single bit adder definition before uniquifying, the modifications will apply to all four adders. If we instead uniquify first, then we can easily modify only the adder of interest.

The uniquify algorithm included in SpyDrNet makes every definition in the entire netlist unique. An interesting area for future exploration is to uniquify components as needed. This would involve only creating a unique definition for the single component that is being modified and allowing other unmodified components to continue to share definitions.

The following code example shows uniquify being used in SpyDrNet.

```python
import spydrnet as sdn
from sdn.uniquify import uniquify

netlist = sdn.load_example_netlist_by_name('fourBitCounter')
```
6.3 Redundancy and Reliability

One method of increasing the reliability of a circuit is to apply redundancy. When combined with repair, this can help increase the reliability of a circuit in the long term. This is frequently accomplished with n-modular redundancy where $n$ is the number of redundant copies of each component. Another related application is duplicating the circuit and comparing the output to alert the user to a discrepancy. This is called duplication with compare (DWC). Both reliability applications have been built on top of SpyDrNet [6], [7].

6.3.1 TMR

TMR is one method by which circuits can be made more reliable. TMR triplicates portions of the circuit to allow the circuit to continue to provide the correct result even under some cases of error. Voters are inserted between triplicated circuit components to pass the most common result on to the next stage of the circuit [19]. Figure 6.1 shows two typical layouts for TMR. The top half of the image shows a triplicated circuit with a single voter that feeds into the next stage of the circuit. The bottom of the figure shows a triplicated voter layout such that even a single voter failure may be tolerated.

Custom code has been written implementing TMR based on the SpyDrNet framework. It can replicate subsets of the circuit, insert the voter logic between triplicated layers, add reduction voting, and connect the triplicated logic in place of the original implementation. The TMR algorithm takes advantage of SpyDrNet’s ability to maintain hierarchy through the tool. This allows the triplicated design to take advantage of the benefits of hierarchy including readability and reduced repetition. The triplicated design was programmed to an FPGA after being processed using SpyDrNet.
6.3.2 DWC

DWC is a reliability algorithm in which the user will duplicate components of the design and include comparators on the output to present a flag that will be raised when the circuit’s outputs do not match [18]. Like TMR’s voters, the comparators can be duplicated as well to ensure that if a comparator goes down at least one of the comparators will flag an issue. Figure 6.2 shows an example of DWC with duplicated comparators. Although the comparators break the output lines in the figure, this does not need to be the case. DWC needs to just listen to the signals and can have less impact on timing, only increasing the fan-out but not correcting errors.

Like TMR, DWC was implemented on SpyDrNet. Once again this was able to take advantage of SpyDrNet’s hierarchy and maintain that through the modification. Comparators were created and inserted and the selected portion of the design was duplicated. The resulting circuits were programmed to an FPGA after being read into SpyDrNet, modified and written back out. As with TMR the existing implementation on the BYU EDIF Tools [8] required that the design be flattened before being processed. [5]
6.4 Clock Domain Analysis

In hardware various clocks are often used in different portions of the circuit. Sometimes inputs and outputs will come in on a different clock before they reach the main pipeline of the circuit. At the junctions between clock domains, circuitry should not be triplicated in TMR. If it is triplicated it may result in steady state error on the output because the signals from the three inputs may reach the clock crossing at separate times and be registered improperly [25]. This can make the overall reliability of the system lower than it otherwise would be.

To find these locations, clock domains have been examined using SpyDrNet. The basic methodology for doing this was to find the clock ports on the various components in the design which have them and trace those clocks through the netlist. The resulting connected components form a clock domain. When a triplication pass encountered the boundary between domains the triplicated circuit could be reduced to a single signal to cross the boundary [5], [6].

6.5 Graph Analysis and Feedback

While triplicating a design users must determine the best location to insert voters in the design. Voters could be inserted liberally at the cost of the timing of the critical path. Alternatively sparse voter insertion can yield a lower reliability. Several partial triplication methods have been designed on the SpyDrNet framework. One study concluded that inserting voters after high fan-out flip flops in a design yielded good results [20]. Another voter insertion algorithm was implemented on SpyDrNet after doing analysis using NetworkX [3] to find feedback loops [5].

Figure 6.2: Duplication with compare showing the duplicated circuitry and duplicated violation flags [5]
In summary, SpyDrNet has been used to create several applications, encompassing hierarchical netlist restructuring, reliability, and circuit analysis. These applications are useful aids in netlist modification. These applications demonstrate the usefulness of having a tool to operate on structural netlists but, they do not represent the entirety of applications which could be built using SpyDrNet. SpyDrNet would be a good target for applications involving any change or observation at the netlist level.
SpyDrNet is a useful tool for modifying and analyzing circuit structure post synthesis. It is unique because it is the only tool of its kind that the author is aware of that is open source and written in Python. Additionally it provides parsers for both EDIF and Verilog netlists allowing the same modification passes to be performed on circuits from both formats. Its independent intermediate representation provide a strong platform on which future work can be based to apply existing algorithms to new formats or new algorithms to an array of existing formats, vendors, and devices.

SpyDrNet contains the intermediate representation, a generic format that can hold netlists from various source formats. All the constructs necessary to represent these netlists are present and leveraged by the parsers and composers to ensure that source formats from existing tools are compatible with SpyDrNet.

The intermediate representation has constructs for most of the common features in netlists. Features that are not common among netlists can be maintained in the metadata. This makes the intermediate representation flexible enough to represent several formats successfully while still providing enough structure to make application passes useful on each supported format.

SpyDrNet’s API is designed to hold netlist representations so that it can be independently leveraged by parsers and composers. These parsers and composers can be swapped with alternative parsers and composers to handle additional formats. This can provide future formats with the functionality and applications that have already been created for SpyDrNet. This makes SpyDrNet compatible with future formats and applications. New parsers and composers can be created without modifying the API calls or elements.

SpyDrNet has several useful applications and utility functions present that both make it more useful to a wide audience and demonstrate its utility. SpyDrNet is capable of handling hierarchical netlists, flattening or uniquifying those netlists, and cloning arbitrary portions of a netlist.
Applications have been built to analyze clock domains and create redundancy within digital circuits [5]–[7].

This work has created an opportunity for future work to advance digital circuit netlist manipulation. Additional applications can be built, more formats can be supported, and plugins can be designed to aid in circuit manipulation. A comparator and consistency checker are some examples of applications that could be created to help expand the SpyDrNet ecosystem.

As more users migrate toward SpyDrNet and the key components become more well tested and hardened, some of these components could be converted into C++ or another low-level compiled language. This can speed up modifications and further increase the utility of the tool. A wrapper could be maintained to ensure compatibility with existing python applications. This would also ensure future rapid python development around SpyDrNet is a continued possibility.

The callback framework could add a post modification callback to help users divide up callbacks that stop additions and those that rely on the addition being made. Other analysis callbacks might be advantaged by being able to work with an already modified data structure. This modification would add one additional function call and a list size check to the watched API functions without any callbacks being registered.

The namespace manager could also be further improved by adding the ability to populate managed namespaces after the creation of a data structure. This functionality would cause additional overhead at switch but should not have a major impact on the overall complexity of the namespace manager. This change would help users who desire to change the formats from the original.

More research could be done to help understand how existing design tools handle flat netlists when compared with hierarchical netlists. Flat netlists remove some information that could be used by tools to help group components. It is a matter of question how or if existing tools leverage this information in any significant way. Some have speculated that the information presented in hierarchy may actually slow down and hinder the further steps generating final target platform. Further research could be done to analyze this hypothesis.

On top of this further work could be done to make sure that the intermediate representation is more generic for parsing and composing. Parameters could be simplified and included as part of the API for example, to make sure that the intermediate representation is more generic. It would
be useful to add switches to the existing parser and composers to help support different flavors of EDIF and Verilog and allow for composing to different tools on the back-end.

SpyDrNet is a desirable target for users developing parsers, composers, and applications because each of the existing designed components can be immediately applied as support is added. Due to its open-source nature SpyDrNet can be used by a wider audience and promote additional feedback and contributions [2]. SpyDrNet is the foundation on which a wide array of tools and users can build to progress digital netlist modification well into the future.
REFERENCES

[8] B. Y. University, “BYU EDIF tools,” Online, July 2021. 6, 15, 49


APPENDIX A.  CODE EXAMPLES

A.1  Netlist in Various Formats

Two example netlist files were synthesized from a two bit combinatorial adder described in Verilog. The Verilog source describes two single bit adders and concatenates them into a two bit adder. This was done in an attempt to encourage the synthesis tool to implement a hierarchical netlist. Xilinx Vivado 2020.2 was used to synthesize the Verilog source resulting in the Verilog and EDIF netlist files included here.

A.1.1  Verilog Source

```verilog
module add2(
    input [1:0] A,
    input [1:0] B,
    output [1:0] O
);

wire carry;

add1 a1(
    .A(A [0]),
    .B(B [0]),
    .cin (0 'b0),
    .O(0 [0]),
    .cout (carry)
);

add1 a2(
    .A(A [1]),
    .B(B [1]),
```
module add1(
   input A,
   input B,
   input cin,
   output O,
   output cout
);
    assign {cout, O} = A + B + cin;
endmodule

A.1.2 EDIF formatted netlist

(edif add2
   (edifversion 2 0 0)
   (edifLevel 0)
   (keywordmap (keywordlevel 0))
   (status
      (written
         (timeStamp 2021 09 07 01 52 29)
         (program "Vivado" (version "2020.2"))
         (comment "Built on 'Wed Nov 18 09:12:47 MST 2020'")
         (comment "Built by 'xbuild'")
      )
   )
   (Library hdi_primitives
      (edifLevel 0)
      (technology (numberDefinition ))
      (cell IBUF (celltype GENERIC)
(view netlist (viewtype NETLIST)
  (interface
    (port O (direction OUTPUT))
    (port I (direction INPUT))
  )
)

(cell OBUF (celltype GENERIC)
  (view netlist (viewtype NETLIST)
    (interface
      (port O (direction OUTPUT))
      (port I (direction INPUT))
    )
  )
)

(cell LUT2 (celltype GENERIC)
  (view netlist (viewtype NETLIST)
    (interface
      (port O (direction OUTPUT))
      (port I0 (direction INPUT))
      (port I1 (direction INPUT))
    )
  )
)

(cell LUT4 (celltype GENERIC)
  (view netlist (viewtype NETLIST)
    (interface
      (port O (direction OUTPUT))
      (port I0 (direction INPUT))
      (port I1 (direction INPUT))
      (port I2 (direction INPUT))
      (port I3 (direction INPUT))
    )
  )
)

(cell INV (celltype GENERIC)
(view netlist (viewtype NETLIST)
   (interface
      (port I (direction INPUT))
      (port O (direction OUTPUT))
   )
)(Library work
   (edifLevel 0)
   (technology (numberDefinition ))
   (cell add1 (celltype GENERIC)
   (view add1 (viewtype NETLIST)
      (interface
         (port (rename A_IBUF_0_ " A_IBUF [0] ") (direction INPUT))
         (port (rename B_IBUF_0_ " B_IBUF [0] ") (direction INPUT))
         (port (rename 0_OBUF_0_ " 0_OBUF[0] ") (direction OUTPUT))
      )
      (contents
         (instance 00 (viewref netlist
            (cellref LUT2 (libraryref hdi_primitives)))
            (property INIT (string "4'h6")))
         )
         (net (rename A_IBUF_0_ " A_IBUF[0] ") (joined
            (portref I0 (instanceref 00))
            (portref A_IBUF_0_)
         )
         )
         (net (rename B_IBUF_0_ " B_IBUF[0] ") (joined
            (portref I1 (instanceref 00))
            (portref B_IBUF_0_)
         )
         )
         (net (rename 0_OBUF_0_ " 0_OBUF[0] ") (joined
            (portref 0 (instanceref 00))
            (portref 0_OBUF_0_)
         )
         )
   )
)
(cell add1_0 (celltype GENERIC)

(view add1_0 (viewtype NETLIST)

  (interface
    (port (array (rename A_IBUF "A_IBUF[1:0]") 2)
      (direction INPUT))
    (port (array (rename B_IBUF "B_IBUF[1:0]") 2)
      (direction INPUT))
    (port (rename O_OBUF_0_ "O_OBUF[0]"

      (direction OUTPUT))

  )

  (contents

    (instance 00 (viewref netlist
      (cellref LUT4 (libraryref hdi_primitives)))

      (property INIT (string "16'h9666")))

    )

    (net (rename A_IBUF_0_ "A_IBUF[0]")) (joined

      (portref I3 (instanceref 00))

      (portref (member A_IBUF 1))

    )

    )

  (net (rename A_IBUF_1_ "A_IBUF[1]")) (joined

    (portref I0 (instanceref 00))

    (portref (member A_IBUF 0))

  )

  )

  (net (rename B_IBUF_0_ "B_IBUF[0]")) (joined

    (portref I2 (instanceref 00))

    (portref (member B_IBUF 1))

  )

  )

  (net (rename B_IBUF_1_ "B_IBUF[1]")) (joined
(portref I1 (instanceref 00))
(portref (member B_IBUF 0))
)

(net (rename O_OBUF_0_ "O_OBUF[0]") (joined
(portref 0 (instanceref 00))
(portref 0_OBUF_0_)
)
)

(property ORIG_REF_NAME (string "add1"))
)
)

(cell add2 (celltype GENERIC)
(view add2 (viewtype NETLIST)
(interface
(port (array (rename A "A[1:0]") 2) (direction INPUT))
(port (array (rename B "B[1:0]") 2) (direction INPUT))
(port (array (rename O "O[1:0]") 2) (direction OUTPUT))
)
(contents
(instance
(rename A_IBUF_0__inst "A_IBUF[0]_inst")
(viewref netlist
 (cellref IBUF
  (libraryref hdi_primitives))))
)
(instance
(rename A_IBUF_1__inst "A_IBUF[1]_inst")
(viewref netlist
 (cellref IBUF
  (libraryref hdi_primitives))))
)
(instance
(rename B_IBUF_0__inst "B_IBUF[0]_inst")
(viewref netlist
 (cellref IBUF

(libraryref hdi_primitives)))

(instance
  (rename B_IBUF_1__inst "B_IBUF[1]_inst")
  (viewref netlist
    (cellref IBUF
      (libraryref hdi_primitives)))
)

(instance
  (rename 0_OBUF_0__inst "0_OBUF[0]_inst")
  (viewref netlist
    (cellref OBUF
      (libraryref hdi_primitives)))
)

(instance
  (rename 0_OBUF_1__inst "0_OBUF[1]_inst")
  (viewref netlist
    (cellref OBUF
      (libraryref hdi_primitives)))
)

(instance a1 (viewref add1
  (cellref add1 (libraryref work))))

(instance a2 (viewref add1_0
  (cellref add1_0 (libraryref work))))

(net (rename A_0_ "A[0]") (joined
  (portref I (instanceref A_IBUF_0__inst))
  (portref (member A 1))
)

(net (rename A_1_ "A[1]") (joined
  (portref I (instanceref A_IBUF_1__inst))
  (portref (member A 0))
)

(net (rename A_IBUF_0_ "A_IBUF[0]") (joined
  (portref A_IBUF_0_ (instanceref a1))
  (portref (member A_IBUF 1) (instanceref a2))
  (portref 0 (instanceref A_IBUF_0__inst))
)

)
(net (rename A_IBUF_1_ "A_IBUF[1]") (joined
  (portref (member A_IBUF 0) (instanceref a2))
  (portref 0 (instanceref A_IBUF_1__inst)))
)


(renamed (rename B_0_ "B[0]") (joined
  (portref I (instanceref B_IBUF_0__inst))
  (portref (member B 1))
)
)


(renamed (rename B_1_ "B[1]") (joined
  (portref I (instanceref B_IBUF_1__inst))
  (portref (member B 0))
)
)


(renamed (rename B_IBUF_0_ "B_IBUF[0]") (joined
  (portref B_IBUF_0_ (instanceref a1))
  (portref (member B_IBUF 1) (instanceref a2))
  (portref 0 (instanceref B_IBUF_0__inst)))
)


(renamed (rename B_IBUF_1_ "B_IBUF[1]") (joined
  (portref (member B_IBUF 0) (instanceref a2))
  (portref 0 (instanceref B_IBUF_1__inst)))
)


(renamed (rename 0_0_ "0[0]") (joined
  (portref 0 (instanceref 0_OBUF_0__inst))
  (portref (member 0 1))
)


(renamed (rename 0_1_ "0[1]") (joined
  (portref 0 (instanceref 0_OBUF_1__inst))
  (portref (member 0 0)))
)

)
(net (rename 0_OBUF_0_ "0_OBUF[0]")) (joined
  (portref I (instanceref 0_OBUF_0__inst))
  (portref 0_OBUF_0_ (instanceref a1))
)

(net (rename 0_OBUF_1_ "0_OBUF[1]")) (joined
  (portref I (instanceref 0_OBUF_1__inst))
  (portref 0_OBUF_0_ (instanceref a2))
)

(comment "Reference To The Cell Of Highest Level")

design add2
  (cellref add2 (libraryref work))
  (property XLNX_PROJ_DIR
    (string "/home/dallin/passthrough"))
  (property part (string "xc7k70t6f56-1"))
)

A.1.3 Verilog Formatted Netlist

`timescale 1 ps / 1 ps
module add1
  (0_OBUF,
   A_IBUF,
   B_IBUF);
output [0:0]0_OBUF;
input [0:0]A_IBUF;
input [0:0]B_IBUF;
wire [0:0] A_IBUF;
wire [0:0] B_IBUF;
wire [0:0] O_OBUF;

LUT2 #(
  .INIT(4'h6))
  .I0(A_IBUF),
  .I1(B_IBUF),
  .O(O_OBUF));
endmodule

(* ORIG_REF_NAME = "add1" *)
module add1_0 ...
  (O_OBUF, ...
  A_IBUF, ...
  B_IBUF);
output [0:0] O_OBUF;
input [1:0] A_IBUF;
input [1:0] B_IBUF;

wire [1:0] A_IBUF;
wire [1:0] B_IBUF;
wire [0:0] O_OBUF;

LUT4 #(
  .INIT(16'h9666))
  .I0(A_IBUF[1]),
  .I1(B_IBUF[1]),
  .I2(B_IBUF[0]),
  .I3(A_IBUF[0]),
  .O(O_OBUF));
endmodule

(* STRUCTURAL_NETLIST = "yes" *)
module add2
  (A,
   B,
   O);
input [1:0] A;
input [1:0] B;
output [1:0] O;

wire [1:0] A;
wire [1:0] A_IBUF;
wire [1:0] B;
wire [1:0] B_IBUF;
wire [1:0] O;
wire [1:0] O_OBUF;

IBUF \A_IBUF [0]_inst
  (.I(A [0]),
   .O(A_IBUF [0]));
IBUF \A_IBUF [1]_inst
  (.I(A [1]),
   .O(A_IBUF [1]));
IBUF \B_IBUF [0]_inst
  (.I(B [0]),
   .O(B_IBUF [0]));
IBUF \B_IBUF [1]_inst
  (.I(B [1]),
   .O(B_IBUF [1]));
OBUF \O_OBUF [0]_inst
  (.I(O_OBUF [0]),
   .O(O [0]));
OBUF \O_OBUF [1]_inst
  (.I(O_OBUF [1]),
   .O(O [1]));
add1 a1
  (.A_IBUF(A_IBUF [0]),
   .B_IBUF(B_IBUF [0]),
A.2 Creating a Netlist

The following code is thanks to Jordi Ramos Chen. It is available on Github under a BSD license [22]. This example illustrates how to create a netlist from the ground up in SpyDrNet.

```python
import spydrnet as sdn

netlist = sdn.Netlist(name='netlist')

# initializing working library
library = netlist.create_library(name='work')

# initializing definition widget
def_widget = library.create_definition(name='widget')
netlist.set_top_instance(def_widget, instance_name='widget')

port_a = def_widget.create_port(name='A', direction=sdn.IN)
port_b = def_widget.create_port(name='B', direction=sdn.IN)
port_c = def_widget.create_port(name='C', direction=sdn.IN)
port_d = def_widget.create_port(name='D', direction=sdn.IN)
port_o = def_widget.create_port(name='O', direction=sdn.OUT)

pin_widget_a = port_a.create_pin()
pin_widget_b = port_b.create_pin()
pin_widget_c = port_c.create_pin()
pin_widget_d = port_d.create_pin()
pin_widget_o = port_o.create_pin()
```
# creating the cables for module widget

cable_a = def_widget.create_cable(name='A')
cable_b = def_widget.create_cable(name='B')
cable_c = def_widget.create_cable(name='C')
cable_d = def_widget.create_cable(name='D')
cable_q_1 = def_widget.create_cable(name='Q1')
cable_q_2 = def_widget.create_cable(name='Q2')
cable_o = def_widget.create_cable(name='O')

# creating the wires for widget

wire_a = cable_a.create_wire()
wire_b = cable_b.create_wire()
wire_c = cable_c.create_wire()
wire_d = cable_d.create_wire()
wire_q_1 = cable_q_1.create_wire()
wire_q_2 = cable_q_2.create_wire()
wire_o = cable_o.create_wire()

# initializing definition AND2

def_and2 = library.create_definition(name='AND2')
port_and2_a = def_and2.create_port(name='A', direction=sdn.IN)
port_and2_b = def_and2.create_port(name='B', direction=sdn.IN)
port_and2_q = def_and2.create_port(name='Q', direction=sdn.OUT)

pin_and2_a = port_and2_a.create_pin()
pin_and2_b = port_and2_b.create_pin()
pin_and2_q = port_and2_q.create_pin()

# create two instances of AND2 which resides in widget

inst_and2_1 = \
    def_widget.create_child(name='and2_1', reference=def_and2)
inst_and2_2 = \
    def_widget.create_child(name='and2_2', reference=def_and2)

# initializing definition OR2
def_or2 = library.create_definition(name='OR2')
port_or2_a = def_or2.create_port(name='A', direction=sdn.IN)
port_or2_b = def_or2.create_port(name='B', direction=sdn.IN)
port_or2_q = def_or2.create_port(name='Q', direction=sdn.OUT)

pin_or2_a = port_or2_a.create_pin()
pin_or2_b = port_or2_b.create_pin()
pin_or2_q = port_or2_q.create_pin()

# create an instance of OR2 which resides in widget
inst_or2 = \
    def_widget.create_child(name='or2', reference=def_or2)

# connect all the pins
wire_a.connect_pin(pin_widget_a)
wire_a.connect_pin(inst_and2_1.pins[pin_and2_a])
wire_b.connect_pin(pin_widget_b)
wire_b.connect_pin(inst_and2_1.pins[pin_and2_b])
wire_q_1.connect_pin(inst_and2_1.pins[pin_and2_q])
wire_c.connect_pin(pin_widget_c)
wire_c.connect_pin(inst_and2_2.pins[pin_and2_a])
wire_d.connect_pin(pin_widget_d)
wire_d.connect_pin(inst_and2_2.pins[pin_and2_b])
wire_q_2.connect_pin(inst_and2_2.pins[pin_and2_q])
wire_q_1.connect_pin(inst_or2.pins[pin_or2_a])
wire_q_2.connect_pin(inst_or2.pins[pin_or2_b])
wire_o.connect_pin(pin_widget_o)
wire_o.connect_pin(inst_or2.pins[pin_or2_q])
sdn.compose(netlist, 'test.edf')
A.3 Traversing A Netlist

The following code is thanks to Jacob Brown. It is available on Github under an BSD license [22]. Several functions are included that illustrate how to traverse a netlist structure for various purposes. Comments above each function describe its purpose.

```python
import spydrnet as sdn

# print the hierarchy of a netlist
def hierarchy(current_instance, indentation=""):  
    print(indentation, current_instance.name,  
         " --instance of", current_instance.reference.name,"--")  
    for child in current_instance.reference.children:  
        hierarchy(child, indentation+" ")

# print a list of all libraries and definitions in a netlist
def libraries_definitions(my_netlist):  
    for library in my_netlist.libraries:  
        definitions = \
            list(definition.name for definition in library.definitions)  
        print("DEFINITIONS IN '",library.name,"':",definitions)

# print the connections in a netlist
def print_connections(current_netlist):  
    print("CONNECTIONS:")  
    for instance in current_netlist.get_instances():  
        print("Instance name: ",instance.name)  
        for pin in instance.pins:  
            IN = "EXTERNAL"  
            OUT = "EXTERNAL"  
            for pin in pin.wire.pins:  
                instance = \
                    list(instance.name for instance in pin.get_instances())  
                for port in pin.get_ports():  
                    # for each pin, get associated port, check the direction  
                    if port.direction is sdn.IN:  
                        if IN is "EXTERNAL":
```
IN = port.name + " of " + str(instance)
else:
    IN = IN + ", " + port.name + " of " + str(instance)
elif port.direction is sdn.OUT:
    if OUT is "EXTERNAL":
        OUT = port.name + " of " + str(instance)
    else:
        OUT = OUT + ", " + port.name + " of " + str(instance)
    print("\t",OUT,"---->",IN)

# print the number of times each primitive is instanced
def instance_count(current_netlist):
    print("Number of times each primitive is instanced:")
    primitives_library = \
        next(netlist.get_libraries("hdi_primitives"),None)
    for primitive in primitives_library.get_definitions():
        count = 0
        for instance in current_netlist.get_instances():
            if primitive.name == instance.reference.name:
                count += 1
            print("\t",primitive.name,": ",count)

netlist = sdn.load_example_netlist_by_name("fourBitCounter")

print("HIERARCHY:")
hierarchy(netlist.top_instance)
libraries_definitions(netlist)
print_connections(netlist)
instance_count(netlist)