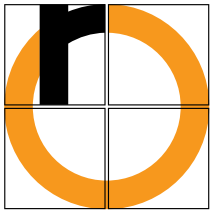
Hochschule **Rosenheim**
University of Applied Sciences



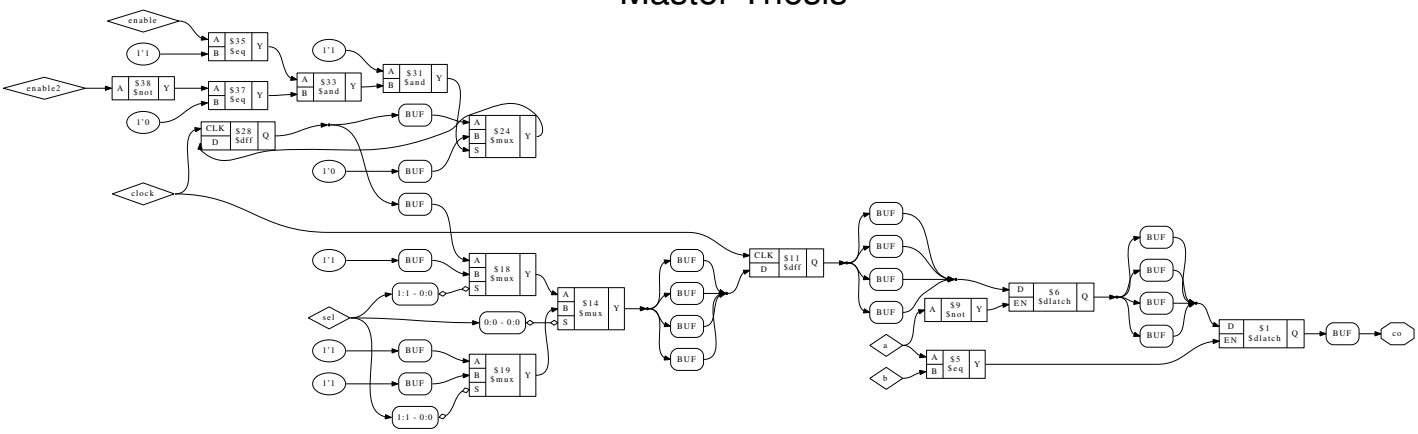
Faculty for informatics

Master course of computer science

Yodl

A VHDL frontend for the open-synthesis toolchain Yosys

Master Thesis



by

Florian Mayer

Date of submission: 15.09.2016
First corrector: Prof. Dr. Ludwig Frank
Second corrector: Prof. Dr. Theodor Tempelmeier

Declaration of authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here.

This paper was not previously presented to another examination board and has not been published.

Rosenheim, 15.09.2016

Florian Mayer

Kurzfassung

Diese Arbeit beschreibt die Entwicklung des ersten open-source VHDL-Frontends mit dem Namen Yodl. Yodl selbst benutzt das quelloffene Synthesetoolkit Yosys als Basis.

Zunächst wird kurz der aktuelle Status der Entwicklung digitaler Schaltungen reflektiert. Außerdem wird Yosys kurz beschrieben. Danach beschreibt die Arbeit erst generelle, compiler-spezifische Probleme, die gelöst werden müssen. Sodann geht sie detailliert auf die VHDL-bezogenen Probleme ein und zeigt die entwickelten Algorithmen und Designentscheidungen auf. Schließlich liefert diese Masterarbeit einen Ausblick in die Zukunft von Yodl, wobei hier sowohl die momentanen Limitierungen der Implementierung als auch Verbesserungen des Testsystems betrachtet werden.

Diese Abschlussarbeit dokumentiert und erklärt den Quellcode von Yodl. Dieser liegt der Arbeit bei und ist gleichzeitig das wichtigste Ergebnis.

Interessant ist sowohl der schriftliche als auch der Quellcodeteil dieser Ausarbeitung für Hardwareentwickler, VHDL-Programmierer, Yosys-Nutzer und Theoretiker aus dem Compilerbau und dem Gebiet der formalen Sprachen. Insbesondere für Theoretiker ist der praktische Aspekt dieser Arbeit relevant.

Schlüsselworte: Compiler, Formale Sprachen, VHDL, Yosys, Hardwaresynthese

Abstract

This work describes the development of the first open-source VHDL frontend named Yodl. Yodl uses the synthesis toolkit Yosys as the foundation.

First of all, the state-of-the-art of digital hardware design is described. Furthermore, some details of the Yosys toolkit are shown. After that, this work illustrates general, compiler-specific problems any translation system faces during implementation. Then, the thesis elaborates on the problems relevant to the processing of VHDL code and shows the appropriate algorithms and design decisions the project uses. Finally, this work provides a glance into the possible future of Yodl where, among other things, current limitations and improvements of the test system play an important role.

This thesis documents and explains the source code of Yodl which was developed in the scope of this master thesis and also represents the greatest achievement of it.

Both, the written part and the source code, might be of interest for hardware developers, VHDL programmers, Yosys users, and theorists with a background on compiler construction or formal languages. For theoreticians, especially the practical aspects of this thesis are likely to be relevant.

Key words: Compiler, formal languages, VHDL, Yosys, hardware synthesis

Contents

1 State-of-the-art of Hardwaredesign **1**
- 1.1 VHDL . . . 1
- 1.2 Yosys . . . 2
- 1.3 Yodl . . . 3

2 Yodl – Subproblems **5**
- 2.1 Lexis and Syntax . . . 5
- 2.2 Compile-time checks . . . 8

3 Yodl – Implementation details **9**
- 3.1 Infrastructure . . . 9
  - 3.1.1 Data model . . . 9
  - 3.1.2 Dot code generator . . . 11
  - 3.1.3 Cloning . . . 16
  - 3.1.4 Generic traverser . . . 17
  - 3.1.5 Type predicates and stateful lambdas . . . 22
  - 3.1.6 Localizing parser data structures . . . 25
  - 3.1.7 Testing . . . 25
- 3.2 AST transformations . . . 26
  - 3.2.1 Loop expansion . . . 26
  - 3.2.2 Generate expansion . . . 29
  - 3.2.3 Elsif elimination . . . 31
  - 3.2.4 If statement elimination . . . 34
  - 3.2.5 Process lifting . . . 35
- 3.3 RTLIL generation . . . 36
  - 3.3.1 Yosys's RTLIL data structures . . . 37
  - 3.3.2 Introduction of SVHDL . . . 37
  - 3.3.3 Synthesis semantics . . . 39
  - 3.3.4 Transformation algorithm – Synthesis examples . . . 44
  - 3.3.5 Transformation algorithm – Implementation details . . . 54
- 3.4 Current Limitations . . . 56

4 Yodl – Future work **57**
- 4.1 Complete parser . . . 57
  - 4.1.1 BNFC and LBNF . . . 57
- 4.2 Further grammar issues . . . 58
- 4.3 Complete VHDL support . . . 59
- 4.4 Far in the future . . . 59
  - 4.4.1 Formal specification of VHDL's synthesis semantics . . . 59
  - 4.4.2 Regression based test suite . . . 59

Bibliography **61**

List of Figures

1.1 Netlist example for a binary adder . . . 3
1.2 Abstraction hierarchy . . . 4
1.3 Main Yosys components and data structures . . . 4

3.1 Class hierarchy of the AST's data model . . . 12
3.2 An example graph generated by Yodl . . . 13
3.3 Runtime structure of a SimpleTree oject . . . 14
3.4 Difference between shallow and deep cloning . . . 17
3.5 Generic traverser runtime behaviour . . . 22
3.6 Important classes and their relationship . . . 38
3.7 Netlist for listing 1.1 generated by Yosys . . . 39
3.8 Netlist for listing 3.37 . . . 48
3.9 Netlist for the listing 3.38 . . . 48
3.10 Netlist for listing 3.39 . . . 48
3.11 Netlist for listing 3.40 . . . 48
3.12 Netlist for listing 3.41 . . . 50
3.13 Netlist for listing 3.42 . . . 53
3.14 Netlist for listing 3.43 . . . 53
3.15 Netlist for listing 3.44 . . . 54

List of Tables

3.1 Truthtable 42

Listings

1.1 Simple full adder in vhdl . . . 2

2.1 Example of a syntactically ambiguous VHDL snippet . . . 7

3.1 Class hierarchy for a simple expression grammar . . . 10
3.2 Representation of $(1+3)*(4+3)$ . . . 10
3.3 Sample snippet for dot graph generator demonstration . . . 11
3.4 A sample graph in dot . . . 15
3.5 Simplified version of emit_vertices . . . 16
3.6 Simplified version of emit_edges . . . 16
3.7 class hierarchy for integer arithmetic expression . . . 17
3.8 Instance of a syntax tree . . . 18
3.9 Eval functions for expression AST . . . 18
3.10 Traverser with Mach7 pattern matching . . . 19
3.11 Interface definition of the GenericTraverser class . . . 20
3.12 Example usage of an GenericTraverser object . . . 21
3.13 An interface for stateful lambdas . . . 23
3.14 N-ary predicate generator . . . 24
3.15 Compile time evaluation of makeNaryTypePredicate . . . 24
3.16 First special case . . . 27
3.17 Second special case . . . 28
3.18 Third special case . . . 28
3.19 Unrolling Scheme – First transformation . . . 29
3.20 Unrolling Scheme – Second transformation . . . 29
3.21 A nested generate statement . . . 30
3.22 Generate statement unrolling . . . 30
3.23 Original if statement with elsif . . . 33
3.24 Desugared if statement . . . 33
3.25 Generated case when statement . . . 34
3.26 Simple concurrent signal assignment . . . 35
3.27 Conditional concurrent signal assignment . . . 35
3.28 Selected concurrent signal assignment . . . 36
3.29 Process encapsulation . . . 36
3.30 A typical IEEE 1076.6 code snippet . . . 40
3.31 Declaration of variables and signals . . . 41
3.32 A typical IEEE 1076.6 code snippet . . . 41
3.33 Simple conditional assignment . . . 43
3.34 Simple D-Latch being utilized . . . 43
3.35 Clock edge specification syntax in VHDL . . . 43
3.36 Public API of NetlistGenerator . . . 45
3.37 Code for a synchronized bit assignment . . . 46

3.38 Code for a nested synchronized bit assignment 46
3.39 Code for a simple latched bit assignment 47
3.40 Code for a nested latched bit assignment 47
3.41 Code for a simple case statement 49
3.42 Code for three nested case statements 49
3.43 Code for a simple synchronized case statement 51
3.44 Code for an if statement actually representing a case statement (aka. muxer) . . 52
3.45 Equal semantics as in 3.44 52
3.46 API of base stack element 54
3.47 Case statement context class derived from stack_element_t 55
3.48 API of base netlist element 55

4.1 Generated classes for expression grammar 57
4.2 Illustration of a common reduce-reduce conflict 58

List of Algorithms

1 Abstract description of a generic traverser's behaviour 20
2 A generic loop pre-processing algorithm 27
3 Generate expansion algorithm 32
4 A simple `elsif` elimination algorithm 33

1 State-of-the-art of Hardwaredesign

Today, almost every part of the work in digital hardware design is done using sophisticated tools and integrated development environments. This chapter will briefly elaborate on how digital hardware is developed nowadays.

1.1 VHDL

The VHSIC Hardware Description Language (VHDL) is a language originally intended for hardware simulation. Its history will not be elaborated here because there already is comprehensive literature about this topic [1].

VHDL quickly evolved from being just a hardware simulation tool to a language that can also be compiled. Since VHDL describes hardware, the target "language" is no language in the sense of x64 or Motorola assembly, but rather a netlist (cyclic graph) of cells connected through wires. Cells can be adders, multipliers or subtraction units and are represented as vertices in the netlist. Wires represent the interconnections between the functional units in the netlist. The term signal vector or signal chunk describes signals with n-bit width. A netlist can be coarsely or finely grained, where a finely grained netlist is obliged to only contain 1-bit input logic gates and 1-bit wide connections. This limitation is not imposed on coarsely grained netlists. Those are even allowed to contain abstractions for n-bit Multiplexers, etc. (cf. [2] chapter 4.2).

To give an intuition of what hardware design with VHDL looks like, listing 1.1 presents a hardware model of a 1-bit wide full adder.

The identifiers a, b, carryIn, carryOut and sum are signals of the top level entity of this model. All those *top-level* signals can have a type, in this case std_logic, and a direction, either `in`, `out` or `inout`. The type std_logic declares all signals to be of 1-bit width. `Entity` declarations describe the interface of a hardware module, whereas `architecture` blocks define the netlist of the previously declared entity.

In `architecture` behv, two statements are present. These are called *signal assignment statements*. The arrow is used to connect two signals together and operators like `xor` or `and` define hardware functionality. Adders would thus be described by the operator +.

Since the code in 1.1 is synthesizeable, a netlist can be compiled from the source. For the model given above, the result is shown in figure 1.1. Note how the semantics of listing 1.1 and graph 1.1 have not changed; illustration 1.1 describes exactly the same data flow as figure 1.1. Only the representation of the hardware description has changed.

In real life, however, hardware designs are much more complicated than in listing 1.1. Because of the incredible complex designs arising in modern hardware industry, VHDL became a viable tool, as it is obvious that the construction of netlists by hand like figure 1.1, does not scale well especially if more than one person is involved in a design project.

Listing 1.1 Simple full adder in vhdl

```
entity adder is
   port(a        : in  std_logic;
        b        : in  std_logic;
        carryIn  : in  std_logic;
        carryOut : out std_logic;
        sum      : out std_logic);
end adder;

architecture behv of adder is
begin
   sum <= a xor b xor carryIn;
   carryOut <=    (a and b)
               or (b and carryIn)
               or (a and carryIn);
end behv;
```

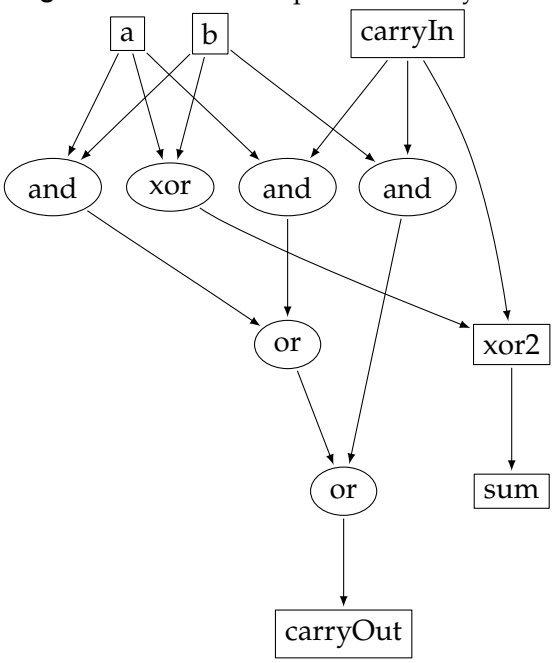
1.2 Yosys

Yodl – the tool described in this work – is based on Yosys. Yosys is an open-source logic synthesis toolkit with various other features. It is actively maintained by Clifford Wolf, who also did almost all of the implementation work (cf. [2]).

Like in so many other topics in science and engineering, different layers of abstraction help to make hardware design automation feasible. Figure 1.2 shows such a layer stack (cf. [2] chapter 2.1). Note that 1.2 is not a comprehensive depiction of those layers. Yosys acts on the abstraction levels *Behavioral Level*, *RTL Synthesis* and *Logic Synthesis*, but it does not implement every algorithm for the transitions between these layers itself. While *Logic Synthesis* can be done entirely using the own logic synthesis algorithms of Yosys, the toolkit also provides an interface to a logic synthesis toolchain called *ABC* which is recommended to be used (cf. [2] chapter 2.1.5 page 16). An in-depth introduction to either one of the depicted levels is beyond the scope of this work. However, chapter 1 of [3] provides an introduction to the topic.

A high-level overview of the architecture of Yosys is necessary, especially for later chapters where Yosys is directly interfaced. Figure 1.3 gives such a birds-eye view and shows how the different components of the toolchain work together. RTLIL and AST are data structures located in memory during the execution of Yosys. RTLIL represents netlists, whereas AST depicts generic abstract syntax trees. Yosys makes it possible to add the VHDL frontend component at two places. First, the frontend could produce an abstract syntax tree using the predefined AST format from Yosys, thus the component would be placed above AST. Second, the VHDL component could emit an RTLIL netlist serialized to normal ASCII text. Said text would then be put into the ilang frontend that does the deserialization. The AST frontend of Yosys is used to convert the AST format into RTLIL netlists. Note that there are various ways to output RTLIL structures. The Verilog backend, for example, generates plain register transfer level Verilog code, whereas the Graphviz backend serves as tool for netlist visualization. Finally, Yosys provides optimizations that act only on RTLIL netlists. Chapter 3.3.1 illustrates the format of those netlists. There are, for instance, optimization passes that eliminate process objects or similar high-level constructs from the netlists and replace them by simple logic cells or wires.

Figure 1.1 Netlist example for a binary adder



1.3 Yodl

Yodl is a standalone program that generates Ilang code. Ilang code, is just another representation of RTLIL from figure 1.3 and as such it is just a netlist. Yodl, in turn, uses VHDL files as input in order to generate those netlists. So, consequently, Yodl transforms VHDL code into RTLIL. The details regarding this transformation will be explained in detail in chapter 3. The same chapter also contains information about the RTLIL.

Yodl is not a production ready synthesis system yet. The current limitations are listed in chapter 3.4. During the development of Yodl, a subset of VHDL has been developed to hide some complexities not regarded in this work. The subset is called SVHDL (S for simple) and is specified briefly and informally in chapter 3.3.2.

As stated in the abstract, a major achievement of this thesis is the implementation of Yodl itself. The entire source code is available on Github: `https://github.com/forflo/yodl`. File names referring to sources in this repository are printed `using this font` and, unless otherwise noted, are relative to `https://github.com/forflo/yodl/tree/master/vhdlpp`.

Figure 1.2 Abstraction hierarchy

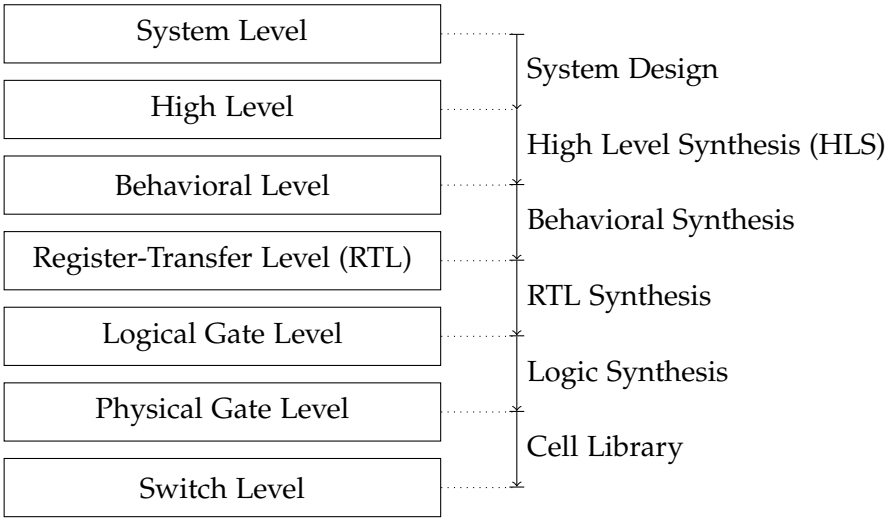
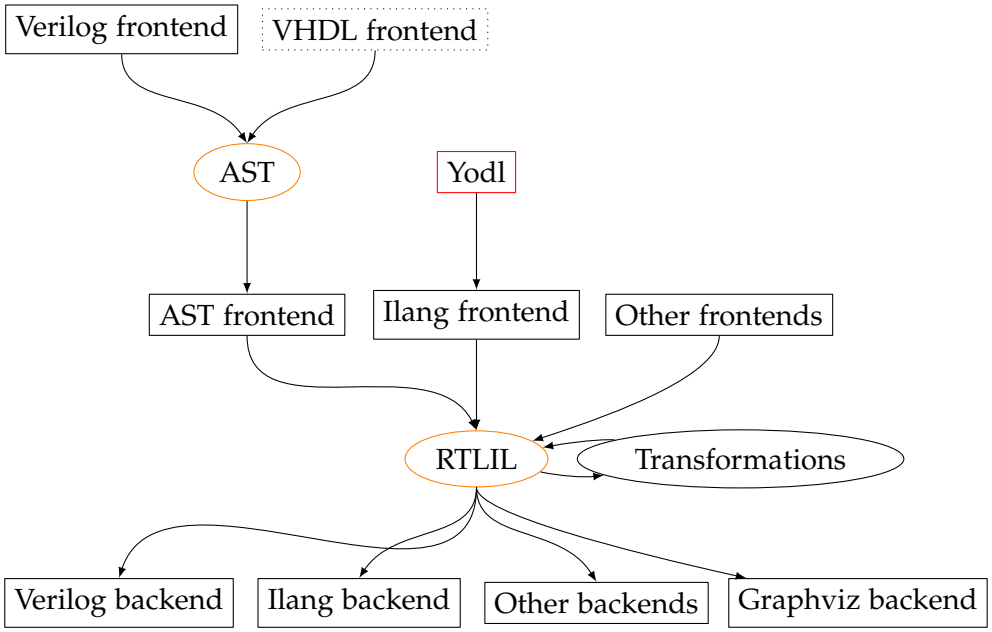


Figure 1.3 Main Yosys components and data structures



2 Yodl – Subproblems

Any compiler construction project can be split up into three distinct subprojects: Lexis and Syntax, validity checks regarding the semantics and code/hardware generation. The last step, hardware generation, is also usually referred to as *synthesis*. The first section of this chapter shows the parsing and lexing problems caused by VHDL's unorthodox grammar. Section two briefly describes the concept of language semantics and answers the question why Yodl can't provide proper compile-time semantics checks. The last part of the problem, synthesis, will not be elaborated here because chapter 3 contains detailed information about this topic.

2.1 Lexis and Syntax

In order to process a programming language of any kind, an abstract syntax tree (AST) has to be constructed. A software component called *parser* usually does that. The parser takes tokens produced by a tokenizer (also called lexer) and incrementally builds said AST. In theory, every programming language can be expressed using only one formal grammar (concrete syntax). However, this grammar would even have to include rules describing how identifiers, numbers or even strings can be constructed, making it very convoluted – and as a consequence – harder to understand. Thus, a two-layer approach is used ever since the very first compilers were built. The first layer solely contains the lexical analysis. In it, every token (parentheses, identifiers, operator symbols ...) is described by a corresponding regular expression. The result of the first step is a sequence of tokens which the parser uses for the second step. This second step consists of a parsing algorithm that can match context-free grammars (mostly of type LALR(1)). The vast set of details bound to formal languages and, in consequence, parsing in general, is beyond the scope of this document and will not be elaborated.

Since the rise of parser generators (such as YACC in the early 1970s) very efficient parsers can be automatically generated from context free grammars encoded in BNF (Backus-Nauer Form). The majority of these code generators put certain restrictions onto the grammars which they can process. Grammars usually have to be in a well-defined subset of the set of the context-free languages; LALR(1) is such a subset. In order for a language to be an element of LALR(1), it must not contain any shift/reduce or reduce/reduce conflicts. This property can be mechanically checked using sophisticated mathematical algorithms (cf. [21]).

VHDL is a special case because its grammar neither is a member of LR(1) – a superclass of LALR(1) – nor LALR(1). To proof this, a part of VHDL's grammar is examined.

⟨*name*⟩ ::= ⟨*simple_name*⟩
| ⟨*operator_symbol*⟩
| ⟨*character_literal*⟩
| ⟨*selected_name*⟩
| ⟨*indexed_name*⟩

| ⟨*slice_name*⟩
| ⟨*attribute_name*⟩

⟨*function_call*⟩ ::= ⟨*name*⟩ ['(' ⟨*association_list*⟩ ')']

⟨*association_list*⟩ ::= ⟨*association_element*⟩ { , ⟨*association_element*⟩ }

⟨*association_element*⟩ ::= [⟨*formal_part*⟩ '=>'] ⟨*actual_part*⟩

⟨*formal_part*⟩ ::= ⟨*name*⟩
| ⟨*name*⟩ '(' ⟨*name*⟩ ')'

⟨*actual_part*⟩ ::= ⟨*actual_designator*⟩
| ⟨*name*⟩ '(' ⟨*actual_designator*⟩ ')'

⟨*expression*⟩ ::= ⟨*name*⟩
| ⟨*number*⟩

⟨*actual_designator*⟩ ::= ['inertial'] ⟨*expression*⟩
| ⟨*name*⟩
| 'open'

⟨*prefix*⟩ ::= ⟨*name*⟩
| ⟨*function_call*⟩

⟨*selected_name*⟩ ::= ⟨*prefix*⟩ '.' ⟨*suffix*⟩

⟨*attribute_name*⟩ ::= ⟨*prefix*⟩ ''' ⟨*attribute_designator*⟩ ['(' ⟨*expression*⟩ ')']

⟨*slice_name*⟩ ::= ⟨*prefix*⟩ '(' ⟨*discrete_range*⟩ ')'

⟨*indexed_name*⟩ ::= ⟨*prefix*⟩ '(' ⟨*expression*⟩ { ',' ⟨*expression*⟩ } ')'

⟨*operator_symbol*⟩ ::= ⟨*string_literal*⟩

⟨*character_literal*⟩ ::= ''' ⟨*graphic_character*⟩ '''

Although this is only a very small subset of the VHDL grammar, it is still hard to see the ambiguity. The following paragraph will present a set of valid simplification steps, that produce subsets of the grammar listed above. The removal of any grammar rules and non-terminal symbols does not create ambiguities but only a new subset of the original language. The idea behind this proof is, to keep deleting non-terminals and grammar rules until it becomes obvious enough to trivially show the existence of a reduce/reduce or shift/reduce conflict. Since only a strict subset of the grammar remains, the proof is sound.

The deletion of the following non-terminals, serves as start.

⟨*operator_symbol*⟩ ::= ⟨*string_literal*⟩

⟨*character_literal*⟩ ::= ''' ⟨*graphic_character*⟩ '''

Since "slice_name" can be expressed by the non-terminal "indexed_name" the following can also be deleted:

⟨*slice_name*⟩ ::= ⟨*prefix*⟩ ‘(’ ⟨*discrete_range*⟩ ‘)’

Also, the rules

⟨*selected_name*⟩ ::= ⟨*prefix*⟩ ‘.’ ⟨*suffix*⟩

⟨*attribute_name*⟩ ::= ⟨*prefix*⟩ ‘’’ ⟨*attribute_designator*⟩ [‘(’ ⟨*expression*⟩ ‘)’]

get deleted.

Furthermore, simplification can be done by defining function calls to be productions of

⟨*function_call*⟩ ::= ⟨*name*⟩ [‘(’ ⟨*expression*⟩ ‘)’]

This produces a language subset, where only unary functions can be called.

After these subsetting steps, only the following grammar remains. Note that *prefix* does not exist because it has been subsumed by *indexed_name*.

⟨*name*⟩ ::= ⟨*simple_name*⟩
| ⟨*indexed_name*⟩

⟨*function_call*⟩ ::= ⟨*name*⟩ [‘(’ ⟨*expression*⟩ ‘)’]

⟨*expression*⟩ ::= ⟨*name*⟩
| ⟨*number*⟩

⟨*indexed_name*⟩ ::= ⟨*name*⟩ ‘(’ ⟨*expression*⟩ { ‘,’ ⟨*expression*⟩ } ‘)’
| ⟨*function_call*⟩ ‘(’ ⟨*expression*⟩ { ‘,’ ⟨*expression*⟩ } ‘)’

The reduce/reduce conflict indeed is obvious now. It shows up clearly if one creates two different production sequences that generate the same sequence of tokens.

⟨*name*⟩ ::= ⟨*indexed_name*⟩ ::= ⟨*name*⟩ ‘(’ ⟨*expression*⟩ ‘)’ ::= ‘foo’ ‘(’ 42 ‘)’

⟨*name*⟩ ::= ⟨*indexed_name*⟩ ::= ⟨*function_call*⟩ ::= ⟨*name*⟩ ‘(’ ⟨*expression*⟩ ‘)’ ::= ‘foo’ ‘(’ 42 ‘)’

Reduce/reduce errors generally imply a bad language design. But as mentioned before, VHDL is a special case. It simply might not be possible to create a conflict free context-free grammar where function parameter lists and array subscriptions both use the same parameter/expression grouping symbol; namely “(” and “)”.

Fortunately, there is a way to resolve this ambiguity. Given the following VHDL snippet 2.1

Listing 2.1 Example of a syntactically ambiguous VHDL snippet

```
entity ent is
   port(A     : out  std_logic_vector(1 downto 0));
end ent;

architecture beh of ent is
   signal foo : std_logic_vector(1 downto 0);
begin
   A(1) <= foo(0);
   A(0) <= foo(1);
end beh;
```

The ambiguity can be resolved with the help of the context. In particular the scoping information can be used to figure out whether a name refers to a variable, a type, or a function. In the above source code, this is easy. Since A is declared to be an output signal and foo is a signal local (i.e. only visible) inside the enclosing architecture and because both signals are of type std_logic_vector, it can be inferred that A(1) clearly describes an array subscription.

In order to generate an unambiguous AST, this information needs to be processed by the parser. To be specific, the parser needs to keep track of a scope stack and all visible identifiers, which makes it a scope aware parser. Because the implementation of such a context sensitive parser would probably be enough to provide material for an entire thesis, an existing VHDL parser will be reused for the purpose presented in this master thesis (cf. 3).

2.2 Compile-time checks

Compile-time checks are also commonly referred to as semantic checks. In formal language theory, there generally are two kinds of semantics that can be defined: The *static* semantics and *dynamic* semantics. Both formalisms use the previously defined abstract syntax to give a formal specification and assume only syntactically correct programs as input. Simply put, static semantics describe properties of a program that can be verified during compile time. Hence the term *static*. The type system of a programming language belongs to this semantic class. In the context of VHDL, for example, the correct usage of packages or libraries must be verifiable during translation time too, thus checks regarding those issues are part of a static semantic specification (cf. introductory chapters of [4] and [5]).

The dynamic semantics of a program is either a rigid description of its behavior during runtime (i.e. how does the program manipulate data) or a specification of the end-product to the translation process (i.e. a netlist).

Unfortunately, IEEE 1076-2008, [6], does not give a formal set of rules using mature mathematical frameworks for semantic specifications. Rather, it provides a language description in English (which is a natural language and not meant to be used as a rigid formal tool).

In 1995, however, the formal specification [10] was published which defines VHDL with different mathematical tools. Since then, sadly there hasn't been made any update for this work whatsoever, making it obsolete as the current standard outdates this work by 13 years.

As a consequence, this work will not present any compile-time consistency/semantic checks.

3 Yodl – Implementation details

This chapter will elaborate on information around the actual implementation of the VHDL frontend Yodl. The first chapter examines the infrastructure that has been built. The following chapter describes the various AST transformation passes. The last chapter finally gives information about the netlist generator algorithm and Yodl's current limitations.

Before the first section of this chapter begins, the overall mode of operation of Yodl must be clarified. As chapter 1.3 briefly states, Yodl transforms plain VHDL code into RTLIL netlists. In order to accomplish this, Yodl first reads in the VHDL source and constructs a tree structure representing the code. The scanner component tokenizes the input code according to the language standard and passes the tokens to the parser, which gradually constructs the parse tree (AST) from the bottom up. Unnecessary information gets discarded during scanning and parsing. For example, parentheses or other structuring tokens need not be contained in the tree representation, because the structure of the tree already provides the same information in a much more usable manner. Chapter 2.1 regards some of the more difficult problems in the scanning and parsing parts.

Once Yodl possesses an AST, it must be semantically checked; though this step is currently omitted. Thereafter, AST simplification happens. Compile-time evaluable expressions get evaluated and replace their originals, loops get unrolled, generate statement evaluation happens and syntax sugar gets replaced by simpler, semantically equivalent language primitives. The order of these simplification steps is not hard-wired and can easily be adjusted. Also, they are elaborated in chapter 3.2.

Finally, Yodl conducts the netlist synthesis and emits the serialized netlist to an IO stream. Yodl currently does not do anything besides AST simplification and netlist generation because of the limitations listed at the end of this chapter. Note that figure 1.3 puts Yodl in the greater context of the Yosys toolchain and shows how frontend and backend are intertwined. Furthermore, section 3.3 documents the creation of netlists.

3.1 Infrastructure

3.1.1 Data model

As 2.1 mentioned, Yodl reuses an existing parser for VHDL. This parser comes from the project *vhdlpp* [11]. Vhdlpp already includes a lexer, a parser and suitable classes that describe the abstract syntax tree.

The data model uses inheritance in order to resemble the nature of the productions in the grammar. The principle used, can be explained best using a simple expression grammar.

$\langle Exp \rangle$ ::= $\langle Exp \rangle$ + $\langle Exp1 \rangle$
| $\langle Exp1 \rangle$;

$\langle Exp1 \rangle$ ::= $\langle Exp1 \rangle$ * $\langle Exp2 \rangle$
| $\langle Exp2 \rangle$;

⟨*Exp2*⟩ ::= ⟨*Integer*⟩
| (⟨*Exp*⟩);

A set of data types can now be defined that might be used to build a typed AST for this grammar. Listing 3.1 shows one possible implementation. Using the data types from listing 3.1, the representation of the expression $1 + 3 * 4 + 3$ is as simple as code 3.2.

Listing 3.1 Class hierarchy for a simple expression grammar

```
class Exp { virtual ~Exp() = default; };

class EAdd : public Exp {
public:
  Exp *exp_1; Exp *exp_2;
  EAdd(Exp *p1, Exp *p2);
};

class EMul : public Exp {
public:
  Exp *exp_1; Exp *exp_2;
  EMul(Exp *p1, Exp *p2);
};

class EInt : public Exp {
public:
  Integer integer_;
  EInt(Integer p1);
};
```

Listing 3.2 Representation of $(1 + 3) * (4 + 3)$

```
Exp *expression = new EMul(
    new EAdd(new EInt(1), new EInt(3)),
    new EAdd(new EInt(4), new EInt(3)));
```

Listing 3.1 contains one base class Exp and three derived classes that inherit from it. Since the destructor function of the base is marked `virtual` and `default`, the inheritance tree is polymorphic and, consequently, the compiler generates a V-Table for each child of Exp. Those V-Tables serve two purposes. First of all, they enable virtual dispatch and Secondly, V-Tables are needed for runtime type analysis of polymorphic objects. After all, the cast operator of C++, `dynamic_cast`, can only do downcasts on polymorphic objects. Type analysis enables one to determine whether a pointer of type Exp points to EAdd or any other specialization of that same base class. In other words, a polymorphic inheritance relation that models a formal grammar, can be used to create typed abstract syntax trees, where the type information of the respective nodes is implicitly available through the V-Tables attached to every object and can be explicitly queried using `dynamic_cast`. The details of queries of this nature are a idiosyncracy of C++ and shall not be elaborated any further here.

The main purpose of the method illustrated in listing 3.1 is therefore the creation of typed ASTs.

Listing 3.2 shows the creation of a typed abstract syntax tree. Once created, the tree begins with an EMul object as root containing two pointers. Both point to a distinct EAdd object which, in turn, also point to two EInt objects respectively. EInt(3), for example, can be reached using the path EMul::exp_1 → EAdd::exp_2.

For reference, figure 3.1 shows the inheritance relation between all relevant classes of the data model of the AST that Yodl uses internally for VHDL code. This model is constructed using the same principles, that the listings 3.1 and 3.2 demonstrate. That means, each actual node can be of a non-virtual class type that is present in the illustrated inheritance tree.

3.1.2 Dot code generator

The simple VHDL frontend prototype Yodl – at the time of this writing – uses 5 different transformation passes that operate on one big data structure; namely the AST. One can see very clearly how important it is to be able to visualize this data structure.

The Graphviz project is a very mature graph rendering software. This software uses a declarative language to describe graphs (and thus trees) and provides a large variety of tools that can understand this formalism. Graph descriptions written in this Graphviz language are also informally called *dot*-graphs [12]. Graphviz was chosen for rendering because it provides the means necessary to convert the AST itself into various graphic formats. The remaining section explains how AST's get converted into dot.

How should the visualized AST look like? Given the VHDL snippet 3.3 the fully rendered graph should look like figure 3.2.

Listing 3.3 Sample snippet for dot graph generator demonstration

```
architecture behv of adder is
   signal result : std_logic_vector(n downto 0);
begin
   -- the 3rd bit should be carry
   result <= ('0' & A) + ('0' & B);
   sum    <= result(n - 1 downto 0);
   carry  <= result(n);
end behv;
```

The implementation splits the problem into three major parts:

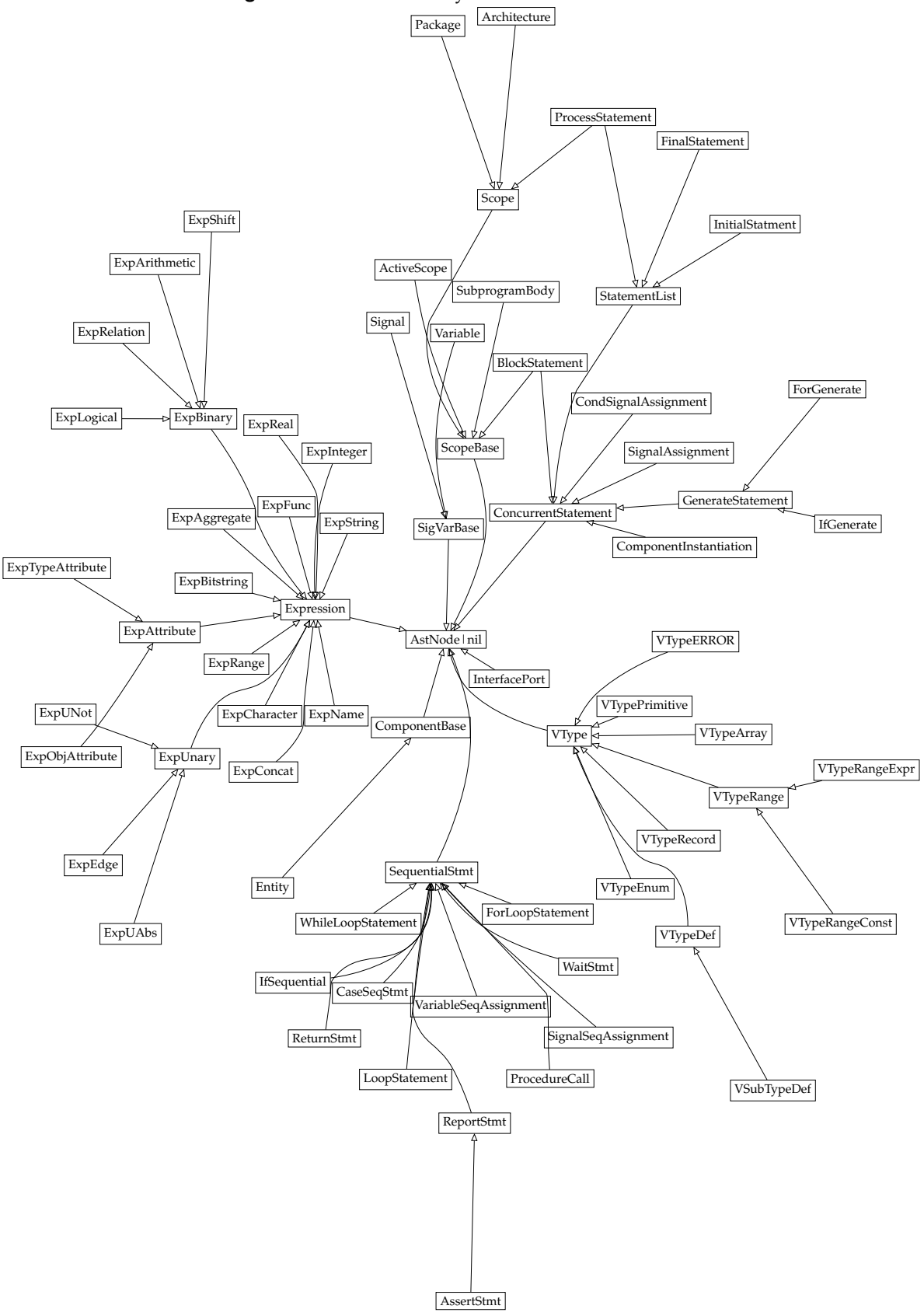
1. Extraction of the relevant information from the AST into an intermediate format
2. Modification of the resulting data for better processing
3. Traversal and code generation

Extraction of information from the AST

After parsing has finished, the complete information from the original VHDL source code lies in the AST. This data structure is built from objects representing expressions, signal assignments or control structures. The current data model, for reference, defines an ExpArithmetic object like the following:

```
class Expression { /* Abstract base class for expressions */ };

class ExpBinary : public Expression {
public:
    ExpBinary(Expression *op1, Expression *op2);
    /* Intentionally left out */

public:
```

Figure 3.1 Class hierarchy of the AST's data model



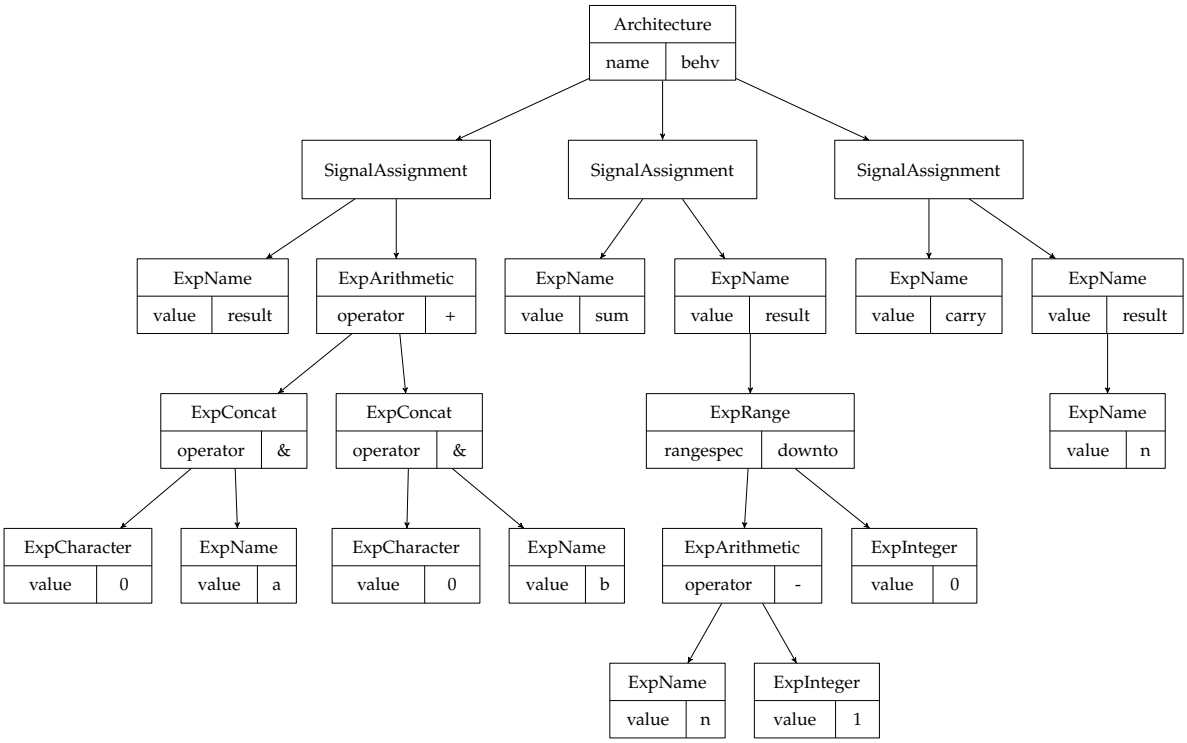


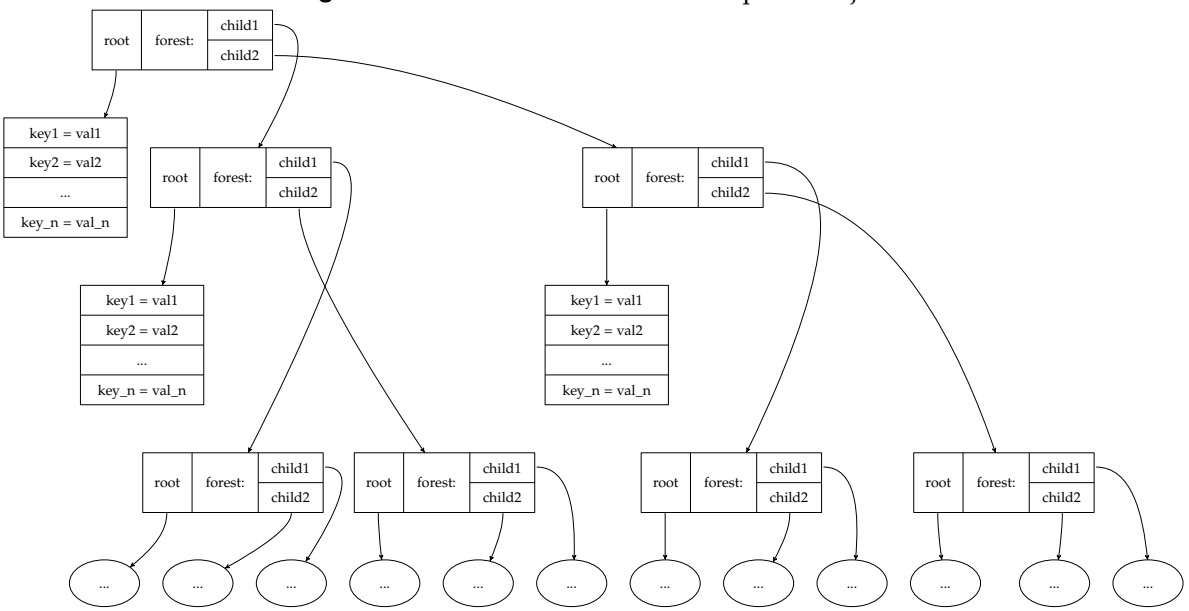
Figure 3.2 An example graph generated by Yodl

```
    Expression *operand1_;
    Expression *operand2_;
};

class ExpArithmetic : public ExpBinary {
public:
    enum fun_t {PLUS, MINUS, MULT, DIV, MOD, REM, POW, xCONCAT};
    ExpArithmetic(ExpArithmetic::fun_t op, Expression *op1, Expression
        *op2);
    /* Intentionally left out */

public:
    fun_t fun_;
}
```

This depiction is heavily simplified and does not comprehensively represent all details of ExpArithmetic objects. Objects that represent whole Architectures are obviously even more complicated. The class definition of an `Architecture` object contains the name of the architecture, a linked list of concurrent statements and all possible declarations that can occur in the architecture header.

In order to extract the mentioned information from the AST, each node (= object) has to be visited. In OOP, there generally are two possible solutions for this kind of traversal. On the one hand, every node object of the AST could implement a virtual method that outputs the desired information in a special data structure. On the other hand, the AST could completely be traversed from outside. However, the second option clearly imposes the public member access upon all member variables of the objects. Within the scope of this work, the first

Figure 3.3 Runtime structure of a SimpleTree oject



solution has been implemented and shall now be elaborated.

As mentioned before, the first step produces data in some kind of intermediate format. Since the AST itself is an n-ary tree, the output ought to be a tree as well. For this purpose, the class *SimpleTree* has been introduced. Its class declaration shall be given in the following listing.

```
class SimpleTree {
public:
    SimpleTree(const std::map<string, string> s) : root(s) { };
    SimpleTree(const map<string, string> s,
               std::vector<SimpleTree<std::map<string,string>>*> own)
        : root(s), forest(own) { };
    /* further ctor and dtor declarations intentionally left out */
public:
    std::map<string, string> root;
    std::vector<SimpleTree<std::map<string, string>>*> forest;
};
```

Note, that the original class uses C++-Templates but, due to the great negative influence of templates on the readability, an instantiated version of the class was printed. As the code shows, the data type used for this instantiation is std::map<string, string>.

In order to visualize the runtime structure of a SimpleTree tree, an example is given in figure 3.3. Now one can take a look at how the mentioned methods inside the AST nodes are implemented. The scheme is as follows: Every AST object inherits the virtual method with the signature

```
SimpleTree<map<string, string>> *emit_strinfo_tree(void) const;
```

Every non-abstract object must provide for a specific implementation that transforms each actual object, `this`, and each of it's successors into a pointer to SimpleTree<map<string, string>>.

For AST nodes that cannot have childs, this implementation is very simple, because it just consists of a statement like

```
return new SimpleTree<map<string, string>>(
    map<string, string>{
        {"node-type", "ExpInteger"},
        /* intentionally left out */,
        {"value", (dynamic_cast<stringstream&>(
                    stringstream{} << value_)).str()}});
```

The C++ statement above produces a SimpleTree with an empty set of successor SimpleTrees and a map containing the type of the object (here ExpInteger), its pointer and its value; where every value is expressed as a string.

For elements inside the AST, the same implementation looks a bit more complicated:

```
SimpleTree<map<string, string>> *ExpRelation::emit_strinfo_tree() const {
    auto result = new SimpleTree<map<string, string>>(
        /* intentionally left out. Analogous to above snipped */);

    result->forest = {
        operand1_->emit_strinfo_tree(),
        operand2_->emit_strinfo_tree()};

    return result;
}
```

Because every ExpRelation object represents a relational operator in the AST, it must also contain two pointers to the operands. In addition, every operand pointer has to be able to point to arbitrary Expressions; hence both operand1_ and operand2_ are of type * Expression;

Modification of the resulting data for better processing

Before the final dot code generation can start, the tree containing the relevant information needs to be modified because of the way the dot language works. In dot, every node must have its own unique identifier, because nodes are implicitly created if a new unknown id appears in the source code. Listing 3.4 illustrates this and also shows how the occurring nodes will be labeled in the rendered picture.

The fact that every node's id has to be unique is problematic, because an AST for the expression $1 + 2 * 3 + 4$ contains 4 ExpInteger and 3 ExpArithmetic objects. As figure 3.2 shows, we want to appear those operator objects in the rendered graph labeled with their type (i.e. ExpArithmetic and ExpInteger). Thus, it's simply not possible to use the type of the node (in the SimpleTree) as node id in the dot code. For that reason, the intermediate tree needs to be augmented with pre-calculated node id's.

The algorithm for this task, however, is not relevant here, but can be viewed in file `generate_graph.cc` at line 77.

Listing 3.4 A sample graph in dot

```
digraph c {
  // two nodes are created: nodeB and nodeA
  nodeA -> nodeB; // nodeB is labeled "nodeB" in rendered graph,
  nodeA [label="foo"]; //whereas "nodeA" is labeled "foo"
}
```

Traversal and code generation

The dot language permits the separation of the node declarations and the specification of their interconnections. Hence, the first traversal only emits all nodes to be connected, whereas the second pass generates the code necessary for the connections between those nodes. The code excerpts from file `generate_graph.cc`, listings 3.6 and 3.5, illustrate the traversal. Note, that the two functions have been heavily simplified for reasons of clarity.

Listing 3.5 Simplified version of emit_vertices

```
void emit_vertices(ostream &out,
                   SimpleTree<map<string, string>> *ast,
                   int depth){
    out << ast->root[NODEID]  << " [label=\""
        << ast->root["label"] << "\"]";

    // recurse into all child trees:
    for (auto &i : ast->forest)
        emit_vertices(out, i, ++depth);
}
```

Listing 3.6 Simplified version of emit_edges

```
void emit_edges(ostream &out,
                SimpleTree<map<string, string>> * ast){
    for (auto &i : ast->forest){
        out << ast->root[NODEID] << " -> "
            << i->root[NODEID]   << ";\n";
        // recurse into all child trees
        emit_edges(out, i);
    }
}
```

The code in 3.5 will run first and emits all node identifiers. Remember, that each SimpleTree node possesses a map, ast->root, containing all relevant attributes. The same goes for listing 3.6.

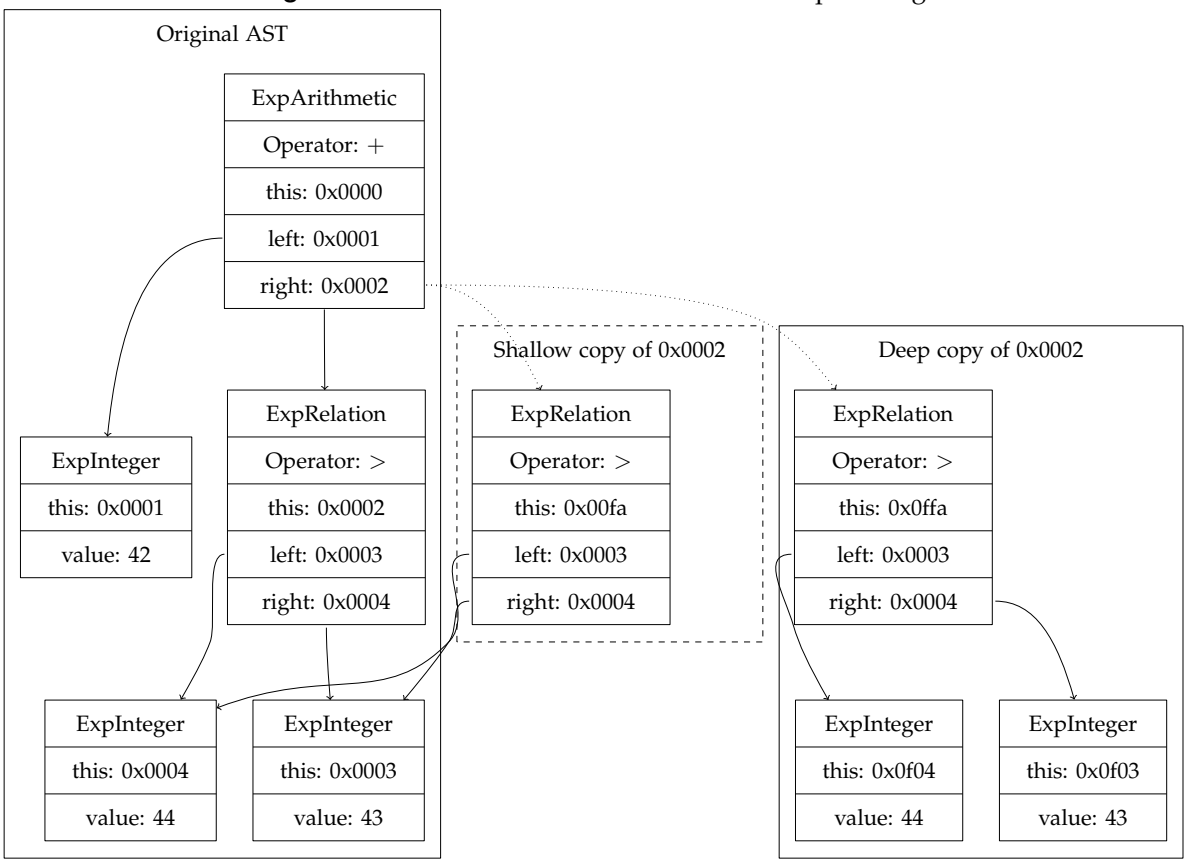
3.1.3 Cloning

Some transformation steps expand certain parts of the AST rather than shrinking it. For instance, loop expansion or generate expansion eliminate looping structures at compile time and replace the affected parts of the AST by parameterized copies of the statements enclosed by the said control blocks.

For this reason, every part of the syntax tree must be deep copyable. The difference between deep and shallow cloning is shown in figure 3.4.

The graphic 3.4 shows that a shallow copy of object 0x0002 only adds one new object to the memory. If the complete subtree beginning at 0x0002 gets freed, the user of the AST can not execute the destructor of the shallowly cloned object, since it contains already invalidated pointers to non-existing objects. Every AST nodes' destructors are recursive. That means that every object, held by the node that the destructor has been called at, gets destructed too. The cloned instance of ExpLogical instead is a full clone, which means that the previously described freeing problem does not need to be considered during its further usage. As a consequence, the code that modifies the AST gets significantly easier to understand.

Figure 3.4 Difference between shallow and deep cloning



3.1.4 Generic traverser

Traversal and evaluation

Due to the recursive nature of an AST, its traversal plays a key role in every compiler development. Arithmetic expressions, for instance, could be modeled by 3.7 (see also section 3.1.1).

Listing 3.7 class hierarchy for integer arithmetic expression

```
struct Exp { virtual ~Exp() {} };
struct Value  : Exp {
    int value; Value (int v) : value(v) {}
};

struct Plus   : Exp {
    Exp* left; Exp* right;
    Plus  (Exp* l, Exp* r) : left(l), right(r) {}
};

struct Minus  : Exp {
    Exp* left; Exp* right;
    Minus (Exp* l, Exp* r) : left(l), right(r) {}
};
```

```
struct Times  : Exp {
    Exp* left; Exp* right;
    Times (Exp* l, Exp* r) : left(l), right(r) {}
};

struct Divide : Exp {
    Exp* left; Exp* right;
    Divide(Exp* l, Exp* r) : left(l), right(r) {}
};
```

Listing 3.8 Instance of a syntax tree

```
Times *exp = new Minus(
  new Plus(
    new Times(new Value(2), new Value(3)),
    new Times(new Value(4), new Value(5))),
  new Divide(new Value(12), new Value(3))
);
```

Using the classes from listing 3.7, the arithmetic Expression $2 \cdot 3 + 4 \cdot 5 - (12/3)$ may be represented by listing 3.8, which is exactly how the parser itself constructs a syntax tree during its reduction phase!

There are two main approaches how arithmetic expressions like 3.8 could be evaluated. Because of the importance of these techniques, both will be elaborated in detail further below. The two evaluation methods are:

1. OOP-like evaluation using member function traversal
2. Functional-style evaluation using an external traverser

Evaluation with 1 requires additional member functions in each of the classes of 3.7 (see snippet 3.9).

Listing 3.9 Eval functions for expression AST

```
// additional function in struct Exp:
    virtual int evaluate() = 0;

/* Further function declarations in structs Plus, Minus
   Divide, Times and Value intentionally left out */
int Plus::evaluate()  { return left->evaluate()+right->evaluate(); }
int Minus::evaluate() { return left->evaluate()-right->evaluate(); }
int Times::evaluate() { return left->evaluate()*right->evaluate(); }
int Divide::evaluate(){ return left->evaluate()/right->evaluate(); }
int Value::evaluate() { return this->value;                        }
```

With the new eval member function, the evaluation of the arithmetic expression from 3.8 becomes as easy as exp->evaluate()! However, this evaluation method comes with two major drawbacks. First of all, the evaluate function must redundantly be declared and implemented for each leaf class of the AST hierarchy. In addition, listing 3.9 shows that also the implementation only differs in the respective arithmetic operation ($+$, $-$, $\cdot$ and $/$). Secondly, due to the first point, the technique doesn't scale well. In more complex AST's with more complex evaluation semantics, the complete traversal process – which is specified by the evaluation functions – is spread over each implementation file of every participating

AST class. That in turn, makes changes difficult to manage and bugs hard to find, which is why Yodl makes no use of this traversal scheme.

Note: The vhdlpp transpiler from the IcarusVerilog project uses exactly the same scheme in order to transpile the VHDL AST into an semantically equivalent Verilog source code (cf. [13] file `expression_emit.cc`).

However, there is another way of doing traversals over syntax trees. Method 2 (see enumeration from above) is a technique frequently found in a functional programming context. It uses so called evaluation/traverser functions in order to extract meaning (aka. semantics) from a given syntax tree. In classical denotational semantics an evaluation function is simply a side effect free function that maps an AST onto a mathematical object that represents the value of the evaluated (or executed) abstract syntax tree (cf. [5], in particular 2.2.2). A main characteristic of these functions is that they use *pattern matching* (cf. [5] figure 4.2 for mathematical pattern matching) in order to determine the type of the current node. Based on this type information, the traverser function determines the fitting traversal for the current AST node. C++, unfortunately, does not natively support pattern matching as a language primitive. However, there are at least two popular libraries – Mach7 and SimpleMatch – that implement such functionality (cf. [14], [15]). Both make heavy use of template meta programming and are thus not easy to understand and explain. For this reason the inner workings won't be elaborated here. Also, no introduction to pattern matching will be provided here, because of its wide adoption in the field of computer science. Nevertheless, materials on that concept can be found in [16] (chapter 4.1, "Pattern matching").

For listing 3.7, a traverser (and evaluator) can be built using pattern matching with Mach7 and looks like snippet 3.10.

Listing 3.10 Traverser with Mach7 pattern matching

```
int eval(Exp *expression){
    using namespace mch;
    using namespace std;
    var<int> i;
    var<Exp *> l, r;
    Match(expression){
        Case(C<Value>(i)){ return i; }
        Case(C<Plus>(l,r)){ return eval(l) + eval(r); }
        Case(C<Minus>(l,r)){ return eval(l) - eval(r); }
        Case(C<Times>(l,r)){ return eval(l) * eval(r); }
        Case(C<Divide>(l,r)){ return eval(l) / eval(r); }
        Otherwise(){ std::cout << "error!" << endl; }
    } EndMatch;
}
```

A natural language description of the source line

```
    Case(C<Plus>(l,r)){ return eval(l) + eval(r); }
```

reads as follows: "If the variable expression has the dynamic type *Plus*, then variable l and r will be bound to `dynamic_cast`<Plus> (expression)->left and `dynamic_cast`<Plus>(expression)->left respectively. If the variable does not possess the type Plus, the succeeding line will be executed".

For reasons of clarity, the mandatory binding template specializations have been omitted. Without these specializations The C<> template can not figure out what values it can bind to the appropriate instances of var<Exp *>.

Generic traverser

Especially for AST transformations, it is necessary to find each node for whom a certain predicate holds true. A common example is loop unrolling, which is described in 3.2.1. Loop expansion clearly makes only sense if it's applied on nodes of type ForLoopStatement. For a compiler writer, the need following algorithm 1 arises. *Predicate* is a higher-order function

Algorithm 1 Abstract description of a generic traverser's behaviour

```
rootNode ← parseVHDL().getRoot()
predicate ← (λtype.λnode.node : type)

function TRAVERSE(node: AstNode *, predicate : (x -> bool), functor : (AstNode * -> void))
    if predicate(node) = true then
        functor(node)
    end if
    for ∀i ∈ node.childs do
        if predicate(i) = true then
            functor(i)
        end if
        TRAVERSE(i, predicate, functor)
    end for
end function
```

that takes one type and maps it onto a new function that checks whether it's input matches this type. A predicate could be constructed, for instance, using the expression

$$predicate\ ForLoopStatement = \lambda node.node : ForLoopStatements$$

where ":" means "has type of". For a given node *node* this anonymous function simply checks if *node* has the runtime type of class ForLoopStatement.

The semantics of the traverse function from above shall be given in natural language too: "For any input node n, the function *traverse* first tests if the predicate holds for n. If it does, the function $functor$, given as parameter to *traverse*, gets executed. This function usually transforms the node in some way. After that, *traverse* iterates over all child nodes i of node n and repeats the previous procedure for each i."

The class GenericTraverser implements this algorithm for the complete class hierarchy depicted in 3.1 and can be found in `generic_traverser.h`.

Listing 3.11 Interface definition of the GenericTraverser class

```
class GenericTraverser {
public:
    enum recur_t { RECUR, NONRECUR };
    GenericTraverser(
        std::function<bool (const AstNode*)> p,
        std::function<int (AstNode *,
                const std::vector<AstNode *> &)> v,
                recur_t r)
        /* Initializer List : */
        : isMutating(true) , isNary(true)
        , predicate(p) , mutatingNaryVisitorU(v)
```

```
        , recurSpec(r)
        { }

        /* The rest of the class internals
           intentionally omitted. */
};
```

Listing 3.11 shows a simplified version of GenericTraverser's class declaration. The first constructor parameter p resembles a so called type predicate, v is the *visitor* function and finally r is used to specify additional behaviour of the traverser algorithm. The usage of an generic traverser is shown by listing 3.12.

Listing 3.12 Example usage of an GenericTraverser object

```
AstNode *ast = /* parsing intentionally omitted */;

StatefulLambda<int> cnt(
    0, [](const AstNode *, int &env) -> int { env++; return 0; });

GenericTraverser counter(
    makeNaryTypePredicate<ProcessStatement, WaitStmt>(),
    [&cnt](const AstNode *n) -> int { return cnt(n); },
    GenericTraverser::RECUR);

counter(ast);
std::cout << "Number␣of␣process␣and␣wait␣statements:"
          << counter.environment << std::endl;
```

Listing 3.12 uses a generic traverser in order to count all AST nodes with a dynamic type of either ProcessStatement or WaitStmt. It uses two currently unknown infrastructure components:

1. A type predicate generator
2. and a stateful lambda.

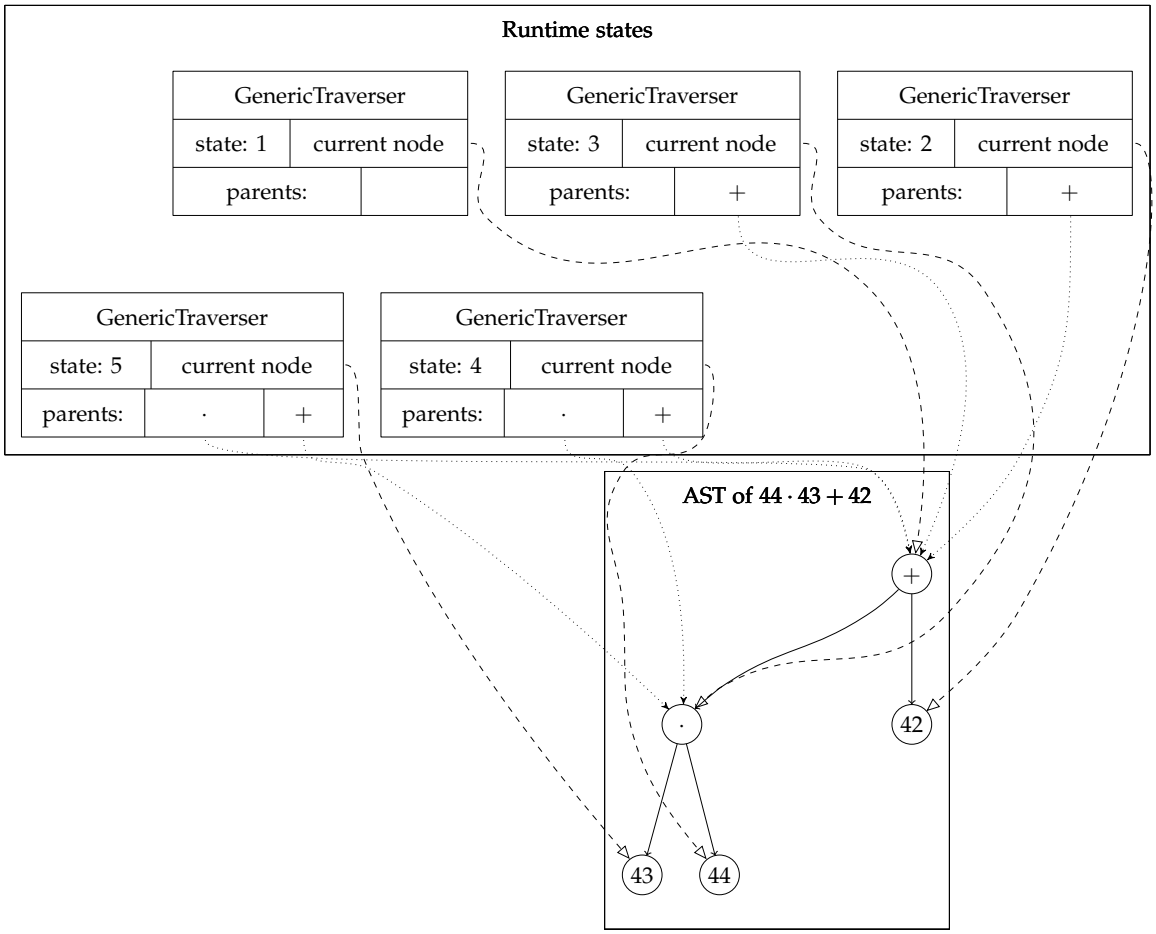
Chapter 3.1.5 describes both, hence a comprehensively explanation is not included in this section. Put simply, a stateful lambda is a C++ object with an overloaded call operator that possesses an internal state that the functor modifies. In the context of 3.12, if cnt's call operator gets executed, it increments its internal state which is a single integer number. In comparison to algorithm 1 the stateful lambda cnt is equal to the parameter *functor* of the *traverser* function. During the recursive, pre-order traversal, a generic traverser keeps track of the current node and all of the nodes ancestors. Figure 3.5 shows that if the traverser visits the leaf node with value 43, the parent vector contains the node $\cdot$ as first and node $+$ as second parent. If, at this state, the requirements of the specified predicate would be met, the visitor

```
std::function<int (AstNode *n,
        const std::vector<AstNode *> &)> v,
        recur_t r)
```

would be called with $n =$ *current node* and $v = [\cdot, +]$.

Internally a generic traverser object manages parent nodes with the help of a stack. Each time it descents further down the abstract syntax tree, it pushes the newly visited node onto the stack and removes it again if the node itself or every child has been visited.

Figure 3.5 Generic traverser runtime behaviour



3.1.5 Type predicates and stateful lambdas

This section presents two minor abstractions that have been proven to be useful for Yodl's implementation. Stateful lambdas are located in `stateful_lambda.h` and type predicates are defined in `predicate_generators.h` and `predicate_generators.cc`.

Stateful lambdas

The generic traverser object uses the constructor shown in 3.11 in order to specify visitor and predicate functions for traversal. This constructor needs a C++ functional with the type

```
std::function<bool (const AstNode*)> p
```

In principle, there are two ways to construct an object of that type. On the one hand, C++-11's lambda syntax can be used as follows:

```
[](AstNode *n) -> int { /* ... */ };
```

On the other hand, the usual and much more verbose way would be to construct a normal C++ functor. In this context, a functor is an object that overloads its call operator appropriately. Let the following code be given as an example:

```
class FunctorTemp {
    FunctorTemp() = default;

    int operator()(/* parameters */){ /* ... */ }
private:
    /* internal state variables here */
};
```

For functors without a persistent internal state, both sources are exactly equivalent. However, lambdas created using the special notation can not contain any private or public member variables (aka. internal state), because there simply is no special syntax for it. At first there seems to be no other way as to stick to the lengthy functor class declaration, but stateful lambdas provide a better solution for simple cases.

The class definition (3.13) for StatefulLambda illustrates how such an improvement can be implemented. Simply put, a StatefulLambda object encapsulates data together with a C++ functor and provides a custom overload for the call operator. That means it can be called like any other function. However, instead of just executing the functor with the parameter passed to `operator()`, it passes along a reference to the internal state environment (see listing 3.13).

Using this new abstraction a functor that counts nodes with particular types can be build easily. Listing 3.12 demonstrates a proper definition of a stateful lambda.

Note, because of clarity reasons, listing 3.13 does not contain all implementation details.

Listing 3.13 An interface for stateful lambdas

```
template<typename T> class StatefulLambda {
public:
    StatefulLambda(T e, std::function<int (AstNode *, T &)> l)
        : environment(e)
        , lambda(l) { }

    // environment gets default initialized in this constructor
    StatefulLambda(std::function<int (AstNode *, T &)> l)
        : lambda(l) { }

    int operator()(AstNode *node){
        return mutatingLambda(node, environment);
    }

    void reset(){
        environment = T();
    }

    T environment;
private:
    std::function<int (AstNode *, T &value)> lambda;
};
```

Type predicates

In order to check whether a given visitor should be applied to a current node during traversal, a generic traverser first executes a previously specified predicate. As mentioned before, if this predicate holds true, the visitor is executed, otherwise it is not.

While visitor functions can either have one or two parameters every predicate must match the visitor signature established in code 3.11:

```
std::function<bool (const AstNode*)>
```

Section 3.1.5 already showed the two main ways to construct these callable objects. However, for type predicates, template meta programming provides the means for an even shorter function constructor. Listing 3.14 illustrates the usage of a predicate constructor. It produces a predicate that maps a pointer or a reference to an object to either true or false depending on whether the dynamic type of the object occurs in the parameter list of the makeNaryTypePredicate template.

Listing 3.14 N-ary predicate generator

```
makeNaryTypePredicate<ProcessStatement, IfSequential>();
```

Every C++11 compatible compiler can handle the meta function (listing 3.14) and expands it in the fashion shown in listing 3.15. The original unexpanded C++ meta code won't be printed here. However, this source code can be examined in `predicate_generators.cc`.

Listing 3.15 shows that the expansion results in an cascaded if-else structure with appropriate Mach7 pattern matches. The matches perform the necessary type checks and return either true if the type matches the specified type or false if this isn't the case. The complexity for the generated function is $O(n)$, with n being the number of template parameters of the makeNaryTypePredicate template function. In the above case (listing 3.14), $n = 2$, because only two type parameters – ProcessStatement and IfSequential – have been used in order to instantiate the function. The reason for the linear time complexity is the progressively cascading if block illustrated by code 3.15. Here, each additional type parameter used for instantiation accounts for an additional nested type test block.

Listing 3.15 Compile time evaluation of makeNaryTypePredicate

```
struct makeNaryTypePredicate {
  bool operator()(const AstNode *n){
    if (helper(n)) {
      return true;
    } else {
      /* anonymous class because of the template recursion */
      struct anon_inner {
        bool operator()(const AstNode *n){
          Match(n){
            CaseInT(mch::C<IfSequential>()){
              return true;
            }
          } EndMatch;
          return false;
        }
      };

      return anon_inner()(n);
    }
```

```
  }

  private:
  bool helper(const AstNode *n){
    Match(n){
      CaseInT(mch::C<ProcessStatement>()){
        return true;
      }
    } EndMatch;
    return false;
  }
};
```

3.1.6 Localizing parser data structures

Yodl uses vhdlpp's AST data model as well as its parser implementation. Vhdlpp uses many global variables and data structures for parsing VHDL (cf. [13] revision 5dd2e6a, file `entity.cc line 27, std_types.cc line 24, 26, ...`). In vhdlpp itself, this does not pose a problem, because the whole program was built with the intention to parse any given source file only once. As a consequence, vhdlpp only produces one AST for each run and exits after it finished its work. Yodl, however, uses smoke tests for verification purposes. Thus, the need for disposable abstract syntax trees arises. Of course, each of these trees could be build by hand, but especially for larger structures this is a tedious and error prone task. For that reason, a new class ParserContext has been introduced and incorporates each of the previous global structures. By means of this new data structure, unit tests can build their own throw-away parser for the use of test AST construction.

The file `parse_context.h` serves as reference for this refactoring effort.

3.1.7 Testing

Compilers are incredibly complex and complicated pieces of software and thus notoriously difficult to test. There are a few well known approaches for compiler validation. The first of which is the so called regression test system...

Recression tests

The data base system *SQLite* for instance, uses such a validation framework (cf. [17]). In order to test sqlite's SQL interpreter, a TCL script randomly generates SQL statements and evaluates them using the interpreter and a predefined data base scheme. The script itself manually calculates the result set and compares it with the output of the SQL interpreter. If both result are exactly equal, the test iteration was successful.

Vloghammer uses a similar concept in order to validate Yosys' Verilog frontend. Akin to SQLite's regression tester, it randomly generates syntactically and semantically correct Verilog code snippets, synthesizes them with Yosys and checks the netlist (cf. [18]).

Such regression testing suites are incredibly useful, but also difficult to implement. Hence, this work won't present an implementation of this kind of automated test frameworks.

Formal verification

The second major compiler validation method uses formal tools. Such validations are very hard to do, because every component of the subjected translator must *mathematically* be proven to work according the the formal specification of the language. That's of course only possible if there exists a formal description of the language's semantics.

There indeed is such a specification, but only for an older version of VHDL, which practically renders the formal verification method unfeasible for VHDL-2008 (cf. [10]).

Smoke tests

So called smoke tests, are much simpler to implement and – if used correctly – can limit the amount of bugs drastically. Tests of this class are usually hand-coded by the compiler writer.

Yodl comes with a set of standard unit tests. At first, the header only library `catch.h` has been used as unit testing framework. Later, a switch to Cpputest was made (cf. [19], [20]). All tests are located in the files `unit_tests_main.cc` and `unit_tests_part1.cc`. Currently there are about 1076 lines of test code.

3.2 AST transformations

All subsections in this chapter describe the various modifications Yodl performs directly on the AST. Each transformation produces an output AST as result. Although they don't directly generate netlists or RTL descriptions, AST transformations like, for example, generate expansion (in 3.2.2), must be performed in order to simplify the AST. Even though netlists could be produced from a raw unsimplified AST in theory, this would be, in practice, completely unfeasible, because of the drastically higher code complexity.

3.2.1 Loop expansion

All looping control structures must be statically unrolled. VHDL describes hardware, that means there is no program counter that might be utilized to implement sequential semantics. Furthermore, since VHDL 2008 (cf. [6]) does not impose any restrictions with regards to the nesting depth of loops, unrolling is no trivial transformation. VHDL-2008 even allows for the usage of `next` and `exit` statements inside of loop bodies. This is a problem, because loops containing these control statements are expected to behave like loops implemented in an imperative and sequential programming language like C. But like mentioned before, bare netlists don't contain primitive operations like conditional branching (as provided by any assembly language).

VHDL's synthesis standard from 2004 (cf. [7]) explicitly forbids the use of `while`-loops. However, in principle, while loops containing arbitrary control expression, *can* be synthesized (cf. [8], in particular chapter 3.2.3.2). The high-level synthesis toolkit Legup demonstrates this, because it allows for the synthesis from ANSI C (including while loops) to behavioural-less[1] Verilog. Legup accomplishes this by creating finite state automata from the behavioural control structure (cf. [3], chapter 2.4.3). The algorithms behind these transformations are very complicated and not subject to further examination, at least in this work. Nonetheless, `while` loops in C programs may contain the same two troubling statements as VHDL models;

1 = Verilog without looping, always blocks, etc ...

`next` $\widehat{=}$ `continue` and `exit` $\widehat{=}$ `break`. That shows, that Legup's RTL synthesis algorithms are at general enough to deal with loops containing an arbitrary number of jumping statements.

Prevention of complexity

A very simple AST transformation method for general loop unrolling is given in algorithm 2.

Here, the algorithm transforms every for loop containing an `exit` or `next` statement into an equivalent while loop. As a consequence, the `for` loop unroller doesn't need to care about those jump statements anymore, because after the illustrated AST modification, there won't be any complicated for loops inside the syntax tree anymore.

While this work presents an working for loop unroller, it does not contain anything with regards to while loop synthesis. Like mentioned above, VHDL 2004 does not allow for those loops to be written in synthesizeable VHDL anyway.

Algorithm 2 A generic loop pre-processing algorithm

for $\forall i \in AST_{original} : isForloop(i) \wedge \neg containsForLoop(i)$ **do**
  if $containsNext(i) \vee containsExit(i)$ **then**
    convertToWhile(i)
  else
    unroll(i)
  end if
end for

Special cases

In certain cases, it is, however, possible to statically unroll a given `for` loop even *if* it contains `next` or `exit` clauses. First, the appearance of next sequential statements shall be illuminated. The first case, listing 3.16, is probably the most obvious. Here, the next statement is executed inside a simple if branch whose condition only depends on the loop index variable – that means that it is statically evaluable.

Listing 3.16 First special case

```
for i in (0 to 2) loop
    if (i = 0) then
        next;
    end if;

    foo <= "001" + i;
end loop;

-- unrolls to
-----------------
-- i = 0;
-- i = 1;
foo <= "001" + 1;
-- i = 2;
foo <= "001" + 2;
```

The second special case is almost as obvious as the first one, but only of theoretical interest, because of its lack of usefulness. In the case shown in 3.17, `next` statements only occur at the first level below the for loop. As an example, consider the following snippet:

Listing 3.17 Second special case

```
for i in (0 to 2) loop
    foo <= "001" + i;

    next;
    unreachedStmt1;
    unreachedStmt2;
    -- etc...
end loop;
```

Another interesting situation is depicted in the following listing 3.18. Can this code be statically unrolled? That depends on the properties that <expression> possesses. First of all, it has to be statically evaluable or else it's not possible to determine the appropriate branch at compile time. Second of all, the if condition must not be modified by the statements $ahead_1 \rightarrow ahead_n$. The reason for the second constraint is given later. Under assumption of code 3.18, the source codes 3.19 and 3.20 illustrate the unrolling scheme.

Listing 3.18 Third special case

```
for i in (0 to 2) loop
    if (<expression>) then
        ahead_1;
        ahead_2;
        -- ...
        ahead_n;

        next;
    end if;

    <statement>;
end loop;
```

Listing 3.19 Unrolling Scheme – First transformation

```
for i in (0 to 2) loop
    if (<expression>) then
        ahead_1;
        ahead_2;
        -- ...
        ahead_n;
    end if;

    if (<expression>) then
        next;
    end if;

    <statements>;
end loop;
```

Listing 3.20 Unrolling Scheme – Second transformation

```
-----------
-- i = 0
if (<expression>) then
    ahead_1; ahead_2;
    -- ...
    ahead_n;
end if;

if (<expression>) then
    next;
end if;

<statements>;
-----------
-- i = 1
if (<expression>) then
    ahead_1; ahead_2;
    -- ...
    ahead_n;
end if;

if (<expression>) then
    next;
end if;

<statements>;
-- i = 1
if (<expression>) then
    ahead_1; ahead_2;
    -- ...
    ahead_n;
end if;

if (<expression>) then
    next;
end if;

<statements>;
```

For reasons regarding generality, the expansion step at the end was not completely printed. Since it was stated that <expression> must be statically evaluable, this evaluation needs to actually happen. However, this is omitted here.

3.2.2 Generate expansion

VHDL contains two distinct syntactic domains where statements can occur. The first one is the so called *concurrent* and the second one is the *sequential* domain. Concurrent statements can only occur in the statement part of an architecture body, whereas sequential statements must be part of `process` blocks. Note that `process` blocks itself are concurrent statements, at least from an syntactic point of view (cf. [6], Annex C – Syntax summary).

So called generate statements belong to the syntactic class of concurrent statements. These represent a kind of language aware macro system, because with their help, code can be generated. In this context, *language aware* means that macros are expanded on AST level rather than plain source code (text).

How generate expansion works

Listing 3.21 A nested generate statement

```
architecture behaviour of ForLoop is
   signal result : std_logic_vector(n downto 0);
begin
   gen : for i in 1 to 2 generate
      nested : for j in 1 to 1 + (i - 1) generate
         sum <= i + j + k;
      end generate nested;
   end generate gen;
end architecture;
```

Listing 3.21 gives a short example of a common generate statement and the code from 3.22 illustrates what those statements expand to.

Listing 3.22 Generate statement unrolling

```
architecture behaviour of ForLoop is
   signal result : std_logic_vector(n downto 0);
begin
   gen : block is
      constant i : natural := 1;
   begin
      nested : block is
         constant j : natural := 1;
      begin
         sum <= i + j + k;
      end block;
   end block gen;

   gen : begin block is
      constant i : natural := 2;
   begin
      nested : block is
         constant j : natural := 1;
      begin
         sum <= i + j + k;
      end block nested;

      nested : block is
         constant j : natural := 2;
      begin
         sum <= i + j + k;
      end block nested;
   end begin;
end architecture;
```

The above example shows, that more than one `block` is labeled with the same identifier. This is no error! VHDL-2008 explicitly requests this kind of behavior during generate elaboration (cf. [6], chapter 14.5.3 – Generate statements). Unelaborated generate statements can themselves contain arbitrary declarations. These declarations will be put into the respective declaration part of the expanded `block`. Note, that only `block` or `process` statements are able to create new scopes inside of architecture statement parts.

Presentation of an own generate expansion algorithm

With the use of the GenericTraverser class, it's very easy to implement this algorithm. A generic traverser is used to model the function *traverser* shown above (see algorithm 3, line 27). In order to realize the logic described by *modify* (algorithm 3, line 6), a custom C++ functor is needed. Recall that a functor is just a C++ object with functional semantics. That means it has some internal state – as every other object – and can be called like an ordinary function; which means, that the object's call operator must be overloaded. GenerateExpander is a class that meets these requirements. Since the expander's implementation very much follows the illustrated algorithm, it shall not be elaborated here any further. The curious reader might consult the files `generate_expander.cc` and `generate_expander.h` for further implementation details.

Note that algorithm 3 contains an undefined function in line 14. This function tries to evaluate the expression of the generate statement. This expression can either be an arbitrary condition or a simple range. In both cases, however, the function must be able to evaluate the expression statically. If that's not possible, an error will be reported and no expansion takes place. For an `if` generate statement, *EXPANDGENERATE* simply encapsulates the statements below the generate statement in a newly created block, but only if the generate condition holds true. If the function, on the other hand, encounters a `for` generate statement, the statement list of the generate block will be replicated for each item in the specified expression. After that, every statement list is placed in accordingly created `block`s analogously to the `if` generate expansion.

3.2.3 Elsif elimination

RTLIL generation is difficult, because the problem – the algorithm that generates RTLIL – can not easily be broken up into sub problems. The method, of course, can be split into different functions that do their respective part of traversing over the syntax tree, but those functions still have their logical place in just one class. The more complicated the input gets, the more complicated the RTLIL generator itself will be. Hence, it makes sense to keep the input AST as simple as possible. `Elsif` elimination is one way to accomplish this.

Example elimination

VHDL provides special syntactic sugar inside of branch statements. The general syntax for `if`-`else` clauses is given below (cf. [6]):

⟨*if_statement*⟩ ::= ⟨*if_part*⟩ ⟨*elsif_part*⟩ ⟨*else_part*⟩ '`;`'

⟨*if_part*⟩ ::= [⟨*label*⟩ '`:`'] '`if`' ⟨*expression*⟩ '`then`' ⟨*sequence_of_statements*⟩

⟨*elsif_part*⟩ ::= '`elsif`' ⟨*expression*⟩ '`then`' ⟨*sequence_of_statements*⟩

⟨*else_part*⟩ ::= ['`else`' ⟨*sequence_of_statements*⟩] '`end`' '`if`' [⟨*label*⟩]

Algorithm 3 Generate expansion algorithm

```
1: rootNode ← parseVHDL().getRoot()
2: currentScope ← nil
3: currentEntity ← nil
4: statementAccumulator ← nil
5:
6: function MODIFY(statements : &list<Statement*>)
7:     tmpStmts ← nil
8:     while containsGenerateStmt(statements) do
9:         for ∀i ∈ statements do
10:            if i = NULL then
11:                return NotOK
12:            end if
13:            if isGenerateStatement(i) then
14:                EXPANDGENERATE(i)
15:                tmpStmts ← statementAccumulator
16:                statementAccumulator ← nil
17:            else
18:                tmpStmt ← i
19:            end if
20:        end for
21:        statements ← tmpStmts
22:        tmpStmts ← nil
23:    end while
24:    return OK
25: end function
26:
27: function TRAVERSER(n)
28:    if n : ScopeBase then
29:        currentScope ← n
30:        if n : Architecture then
31:            return MODIFY(dynamic_cast<Architecture*>(n)->statements)
32:        end if
33:        if n : BlockStatement then
34:            return MODIFY(dynamic_cast<BlockStatement*>(n)->concurrent_stmts_)
35:        end if
36:    else if n : Entity then
37:        currentEntity ← n
38:    end if
39:
40:    for ∀i ∈ n.childs() : pointerNotNull(i) do
41:        TRAVERSER(i)
42:    end for
43:
44:    return OK
45: end function
```

This (simplified) grammar allows for the source code 3.23.

Listing 3.23 Original if statement with elsif

```
if (s = "00") then
   op <= "0";
elsif (s = "01") then
   op <= "1";
else
   of <= "0";
end if;
```

Every `elsif` clause can be mechanically desugared into a nested branch statement only consisting of a `if` and (an optional) `else` clause. For the previous code example 3.23, the listing 3.24 demonstrates the process.

Listing 3.24 Desugared if statement

```
if (s = "00") then
   op <= "0";
else
   if (s = "01") then
      op <= "1";
   else
      op <= "0";
   end if;
end if;
```

Elimination Algorithm

Algorithm 4 A simple `elsif` elimination algorithm

```
1:  for ∀n ∈ AST_original do
2:      if n : IfSequential then
3:          eliminateElsif(n :! IfSequential * )
4:      end if
5:  end for
6:
7:  function eliminateElsif(ifStmt : IfSequential*>)
8:      elsifCarry : list<SequentialStmt *> ← ifStmt->elsePart
9:      tmpResult : IfSequential
10:     for ∀i ∈ reverse(ifStmt− > elsifList) do
11:         tmpResult ← new IfSequential(i->condition, i->ifPart, nil, elsifCarry)
12:         elsifCarry ← makeList(tmpResult)
13:     end for
14:     ifStmt.elsifPart ← nil
15:     ifStmt.elsePart ← elsifCarry
16:
17:     return OK
18: end function
```

The algorithm 4 shows type checks using the operator :. On the right-hand side, the name of the type is given, whereas the left-hand side consists of a variable identifier. Similarly, the :!

operator denotes a dynamic type conversion, which is also known as dynamic cast in C++. In assignment contexts, where the assignment operator ← is used, the operator : shows the declared type of the variable subjected to the assignment.

The implementation of algorithm 4 can be found in `elsif_eliminator.cc` and `elsif_eliminator.h`. It is very short and concise with only 38 lines of code.

3.2.4 If statement elimination

Every if-else clause can be algorithmically transformed into an equivalent case-when clause. This is useful for the same reason described in section 3.2.3. The listing 3.24 transforms very easily into listing 3.25.

Listing 3.25 Generated case when statement

```
case (s = "00") is
   when TRUE  => op <= "0";
   when FALSE =>
      case (s = "01") is
         when TRUE  => op <= "1";
         when FALSE => op <= "0";
      end case;
end case;
```

The transition from if-else into case-when clauses can only be performed

- if the original expression from the `if` statement does not contain any clock edge specification (cf. [7], 6.1.2)
- and if the statement lists for the if and else code path both contain exhaustive signal assignments (i.e. every signal from both paths gets at least one assignment).

If the above requirements are met, the transformation produces a semantically equivalent AST, because of the following reasons:

- The conditions, inside of a `case` head, can be any syntactically valid expressions, as long as its evaluation results in something of a scalar[2] or 1-dimensional array type. If it evaluates to an array type, the elements of the array must all have a scalar type (cf. [6], 10.9). Since an `if` statement condition must evaluate to either true or false, the first requirement holds (cf. [6], 10.8), because true maps onto '1' and false to '0'.
- The conditions inside of the case alternative delimiters (`when` <exp> =>) must be either scalar or of one dimensional array type, where the base type of that array has to scalar (cf. [6] 10.9). Since TRUE and FALSE are enumeration literals, they are scalar.
- Neither `if` nor `case` statements impose any restrictions onto the enclosed sequential statements.

The first sentence of this section states that the main reason for this transformation is to make the AST simple enough to enable a less complex AST *to* RTLIL transformation algorithm. This is still valid, because of the fact that such a algorithm, effectively eliminates a whole class of statements without changing the semantics of the AST. There, however, is another reason why this kind of transformation is useful. The RTL intermediate language of Yosys

2 actually only std_logic or std_ulogic are allowed

supports n-ary case statements natively. Hence, the AST *to* RTLIL translator only has to map each `case` branch onto the according branch structure inside the RTLIL data structure. The current netlist translator, developed in the scope of this work, however, only produces simple cells and makes no use of RTLIL's higher abstractions.

The files `ifelse_case_converter.cc` and `ifelse_case_converter.h` specify and implement the shown conversion method from 4.

3.2.5 Process lifting

In VHDL, there are six different kinds of signal assignment statements which are allowed in a statement list of an architecture or a process.

- Concurrent signal assignment statements
  1. Simple concurrent signal assignment
  2. Conditional concurrent signal assignment
  3. Selected concurrent signal assignment
- Sequential signal assignment statements
  1. Simple sequential signal assignment
  2. Conditional sequential signal assignment
  3. Selected sequential signal assignment

It is possible – and required by the standard (cf. [6], 11.4) – to convert all concurrent signal assignment statements to semantically equivalent sequential assignments. In the context of this work, this procedure is called *process lifting*, because of the fact that a ordinary concurrent statement gets lifted into an sequential context; which is, of course, only present inside an `process` block.

First, let there be some examples for the three kinds of concurrent assignments. The listings 3.26, 3.27 and 3.28 also contain a syntactic overview.

Listing 3.26 Simple concurrent signal assignment

```
-- BNF grammar:
-----------------
-- Simple_Concurrent_Signal_Assignment ::=
--     Target "<=" [ "guarded" ] [ Delay_Mechanism ] Waveform ";";
-- Waveform ::= Expression | "null" | ...;
-- Rule for Delay_Mechanism intentionally ommitted

sigVector <= "01001001";
```

Listing 3.27 Conditional concurrent signal assignment

```
-- BNF grammar:
-----------------
-- Concurrent_Conditional_Signal_Assignment ::=
--     Target "<=" [ "guarded" ] [ Delay_Mechanism ]
   Conditional_Waveforms ";";
-- Target ::= Name | Aggregate;
-- Conditional_Waveforms ::= Waveform "when" Condition
```

```
--                              { "else" Waveform "when" Condition } [
    "else" Waveform; ]

sigVector <= "1111" when (input = '0')
                 else "1000" when (input = '1')
                 else "0000";
```

Listing 3.28 Selected concurrent signal assignment

```
-- BNF grammar:
-----------------
-- Concurrent_Selected_Signal_Assignment ::=
--     "with" Expression "select" Target "<="
--     [ Delay_Mechanism ]  { Waveform "when" [Choice] "," } ";" ;

with tmpInteger select sigVector <=
    "0001" when 0 | 1 | 42,
    "0010" when others;
```

The general scheme for appropriate encapsulation is simple on first sight, but as soon as more details show up, it gets a lot more complicated. In order to enclose one of the shown statements in a process block, one simply has to create such a block using a snippet like 3.29.

Listing 3.29 Process encapsulation

```
sampleProc : process(tmpInteger) is
begin
    with tmpInteger select sigVector <=
        "0001" when 0 | 1 | 42,
        "0010" when others;
end process sampleProc;
```

What needs to be put in the sensitivity list, though? [6], chapter 10.2, provides an algorithm that has to be used to fill the sensitivity list. Note, that it makes no semantic difference whether this list of signals is written right after the `process` keyword enclosed in parentheses, or only as the arguments of an additional `wait` statement at the end of the process's sequence of statements (cf. [6], 10.2).

Because of the huge complexity, this work only implements a simplified version of the official algorithm which can be found inside the files `signal_extractor.cc` and `signal_extractor.h`. The actual process lifting methods are encapsulated in the class CsaLifter which is located in `csa_lifter.cc` and `csa_lifter.h` respectively. The exact implementation details are of no relevance for this work and have only been presented for completeness.

3.3 RTLIL generation

The RTLIL generation is a process that transforms a (simplified) VHDL abstract syntax tree into an functionally equivalent netlist. Netlists have briefly been described in chapter 1.1.

The first subsection below will introduce the RTLIL netlist format used by Yosys' synthesis backend, the second specifies a set of requirements of the input syntax tree and the third describes the netlist generator algorithm itself.

3.3.1 Yosys's RTLIL data structures

The term RTLIL stands for *Register Transfer Logic Intermediate Language*. However, it is not solely a formal language for netlists, but also a set of C++ classes specifying an internal representation that is easily interfaceable from within C++ programs. Figure 3.6 shows the most important classes, their members and their relationship with each other.

Any given RTLIL data structure can be serialized to *ilang*, which is the textual form of RTLIL. Of course, any valid ilang file can also be deserialized again and be stored as common C++ data.

The class Design represents the core of any netlist. A Design object is roughly equivalent to a VHDL top-level entity, because it subsumes all participating submodules and provides for the according interconnections between them. Every design can contain arbitrarily many modules, whose purpose it is to hold wires, cells, memories and processes as well as to connect them together. Modules represent connections via the connections member which is simply a list of SigSig objects. SigSig objects are 2-tuples that associate one signal with another. Signals are modelled via the SigSpec class. It must be noted, that every Wire object can be converted to a SigSpec object, but not vice versa.

The enumeration State models every possible value a connection between two cells in the netlist might hold. The last value, marker is only used internally by some optimization passes.

Right at the end of the member list of Module there are still two entries. The first map associates identifiers with Memory objects which are used to model block RAM resources and the second contains Process objects. Verilog and VHDL both offer sequential (behavioural) hardware description. In both languages behavioural modelling is only possible inside of `always` or `process` blocks respectively. RTLIL's Process objects try to emulate a part of those semantics.

Memory and process objects are not relevant, which is the reason for the ellipses in figure 3.6. As figure 1.3 from section 1.2 shows, there exists a backend that converts netlists into equivalent dot graphs. This is very important for debugging purposes. The picture 3.7 contains a graph describing the netlist for the full adder mentioned in section listing 1.1.

3.3.2 Introduction of SVHDL

The step from abstract syntax trees to netlists is complex enough by itself. Because of that, it's only reasonable to keep the input AST as simple as possible. An AST A is said to be simpler as B if

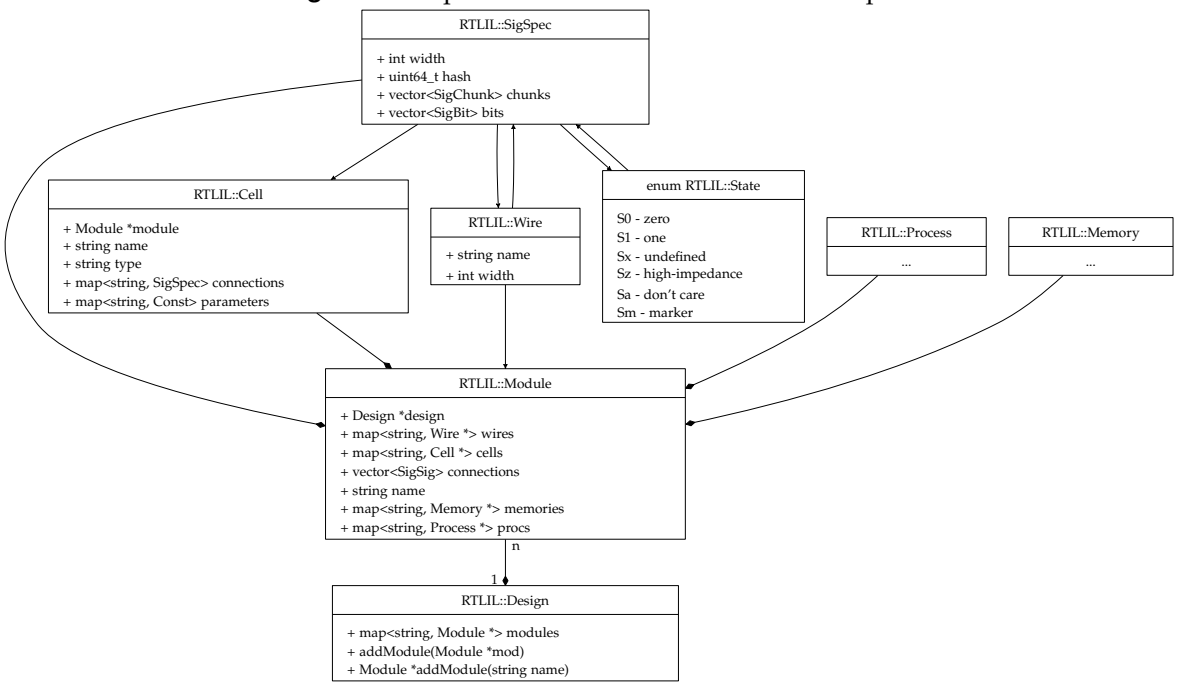
$$\begin{aligned} &\mid t_A \mid < \mid t_B \mid, \; with \\ &t_A = setOfTypes(A) \\ &t_B = setOfTypes(B) \end{aligned}$$

The function $setOfTypes$ maps each syntax tree onto a set of all appearing types the nodes have. An intuitive example: Let there be two different production trees, one for $1 + 2 \cdot 3$ (A) and one for $1 + 2 + 2 + 2$ (B). Now $t_A = \{\mathbb{N}, +, \cdot\}$ and $t_B = \{\mathbb{N}, +\}$ and furthermore $\mid t_B \mid < \mid t_A \mid = 2 < 3$. Thus, B is simpler as A.

Syntax trees are unseparable bound to their associated context free grammars. The more different rules a grammar contains the more different types the equivalent class hierarchy of the object oriented AST incorporates[3]. As a consequence a simpler AST can be specified not

3 This fact won't be proven here

Figure 3.6 Important classes and their relationship



only by restrictions on its actual data structure, but also by restrictions on the context free grammar that generates this abstract tree.

The following BNF code presents the restrictions imposed on (parts of) the original grammar resembling the so called simple VHDL (SVHDL).

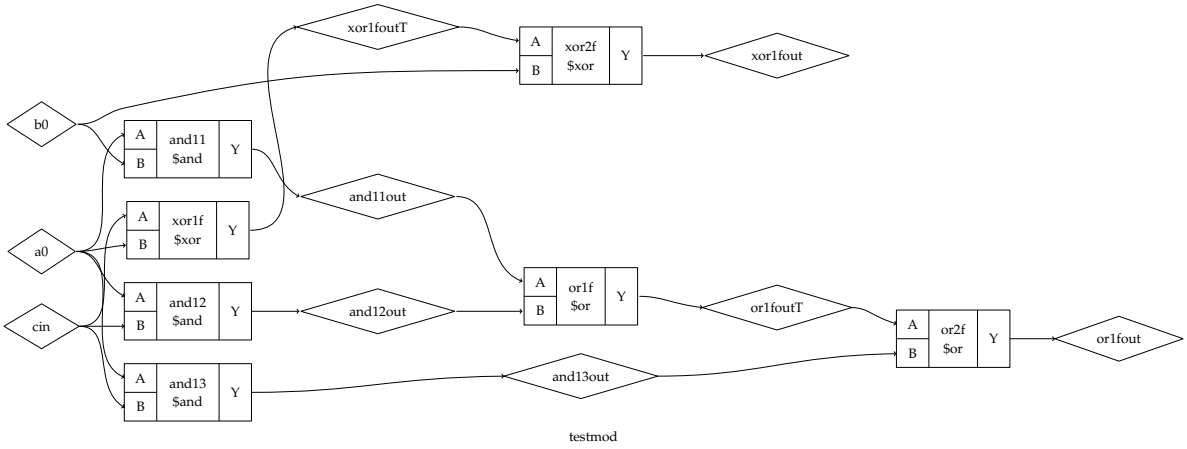
⟨*sequential_statement*⟩ ::= ⟨*wait_statement*⟩
| ⟨*simple_signal_assignment_statement*⟩
| ⟨*simple_variable_assignment_statement*⟩
| ⟨*case_statement*⟩

⟨*concurrent_statement*⟩ ::= ⟨*block_statement*⟩
| ⟨*process_statement*⟩
| ⟨*component_instantiation_statement*⟩

Originally, sequential statements could also be constructed from following statements:

- conditional_signal_assignment
- selected_signal_assignment
- conditional_variable_assignment
- selected_variable_assignment
- if_statement
- loop_statement
- next_statement

Figure 3.7 Netlist for listing 1.1 generated by Yosys



- exit_statement
- procedure_call_statement
- return_statement

Likewise, the concurrent statements rule hat following right-hand sides:

- concurrent_procedure_call_statement
- concurrent_assertion_statement
- concurrent_signal_assignment_statement
- generate_statement

Generate and for loop unrolling eliminates *if_statement* and *generate_statement* as well as *next_statement* and *exit_statement*. Procedure inlining is not part of this work, but would remove *return_statement* and *procedure_call_statement*. The standard for VHDL-2008 shows how *conditional_variable_assignment*, *conditional_signal_assignment*, *selected_variable_assignment* and *selected_signal_assignment* can be transformed in syntax trees only consisting of *if_statement*, *simple_variable_assignment_statement*, *simple_signal_assignment_statement* and *case_statement* (cf. [6], 11.6). Chapter 3.2.4 describes an algorithm with whom it is possible to eliminate *if_statement*. Finally, *concurrent_procedure_call_statement*, *concurrent_assertion_statement* and *concurrent_signal_assignment_statement* can be wrapped in `process` statements and thus be transformed into *sequential_statement* and *process_statement*.

The above elaboration only imposes syntactical restrictions onto VHDL. In order to also specify semantical constraints, one has to use precise mathematical tools, which would easily exceed the scope of this thesis. Consequently, those restrictions will not be given here neither formally nor informally.

3.3.3 Synthesis semantics

In standard [7], a typical semantic specification begins with a listing. Given the snippet 3.30, the standard would describe what hardware must synthesized. It does not specify what algorithm has to be used for this task, as this is not the scope of such a specification.

Listing 3.30 A typical IEEE 1076.6 code snippet

```
AsyncReset: process: (clock, reset)
begin
    if( reset = '1' ) then
        -- async assignment
        Q <= '0';
    elsif( rising_edge(clk) and reset = '0' ) then
        -- sync assignment
        Q <= D;
    end if;
end process;
```

The VHDL code in 3.30 is expected to model a single clock edge sensitive flip-flop with an asynchronous reset behavior. The main characteristic of such a discrete component is fitting for the code 3.30. If reset goes to value 1 the flip-flop overwrites its current content with 0, otherwise the output Q gets set only on occurrences of clock edges (cf. [9], 11.4 – D-Flipflops).

The remainder of this section is concerned with two things: The first subsection will briefly and informally give a definition for the semantics of certain structures and the second subsection will show concrete synthesis results using Yodl.

Semantics of simple assignments

Given the assumption that an appropriate `entity` declaration has already been declared, the code

```
simpleAssign: process(clock, reset)
begin
    foo <= "00011000";
end process;
```

shows a legal signal assignment statements. Note that the enclosing `architecture` has been omitted for reasons regarding simplicity.

Let foo have the array type std_logic_vector(`downto`) with 8 members. Since the base type of such an array is equal to std_logic, the assignment of a string literal becomes possible. In this case, the said string can only contain characters representing the state that a single std_logic value can have (cf. [6], 10.5).

The most important possible values of std_logic are '0', '1', 'z' and '–', where 'z' describes a high-impedance on the associated wire and '–' simply means that the value does not matter and can be anything. '–' signals are not relevant in this work (see figure 3.6).

Consequently, foo <= "00011000" describes that the wires, foo(7) down to foo(0), shall be driven by the respective signal values from the string literal. Hence, only foo(4) and foo(3) are driven with the value '1'.

Semantics of variable assignments

Since, VHDL was originally intended as a language for circuit simulation rather than description, there are a few but important differences between simulation semantics and synthesis semantics. VHDL's reference manual, which describes mainly the simulation semantics, states that all signal assignments encapsulated inside an `process` shall be accumulated until the very end of the process' statement list. If this last statement has been executed, only

then every deferred signal assignment shall be executed at once. Why this semantics was chosen, will not be elaborated any further here. It should be clear, that only by using signal assignments of processes, one can not easily model sequential behavior, because of the previously described semantics. Hence, VHDL's designers introduced the concept of *variables*.

Unlike signal assignments, variable assignments show immediate effect. In other words, if a variable gets updated inside a process using an variable assignment statement, every subsequent usage of the same variable identifier will refer to the right-hand side of the latest assignment.

Listing 3.31 shows how signals and variables have to be declared and how values can be assigned to them.

Listing 3.31 Declaration of variables and signals

```
fooP: process(clock, reset) is
    signal fooS : std_logic_vector(7 downto 0) := "00100111";
    variable fooV : std_logic_vector(7 downto 0) := "00100111";
begin
    -- assignment of a signal
    fooS <= "11111111";
    -- assignment of a variable
    fooV := "11111111";
end process;
```

Synthesis semantics and simulation semantics for variable assignments are the same. However, this is not the case for signal assignments. Due to the lack of time, variable assignments were not implemented in the synthesis algorithm. Consequently, further elaboration is not relevant here. There is, nevertheless, the paragraph 3.3.1.4 in book [8] regarding variable assignment synthesis.

Semantics of Case blocks

Case statements in VHDL are constructed as illustrated in code 3.32. The standard demands case expressions to be either of a scalar type or of one-dimensional array type where the array type's base type must be scalar. The `when` clauses, following the `case` keyword, must be exhaustive. In other words, for each possible value of the case expression there must be one and only one matching choice. If not all choices can reasonably be given – for example if the width of the case expression exceeds 5 bits – a special VHDL keyword must be used: `others` (cf. [6], 10.9).

Listing 3.32 A typical IEEE 1076.6 code snippet

```
architecture b of e is
    signal fnord : std_logic_vector(1 downto 0);
    signal oddParity : std_logic;
begin
    AsyncReset: process(clock, reset, fnord)
    begin
        case fnord is
        -- fnord is called case expression
            when "00" => oddParity <= '1';
            -- when "00" is called a choice
            when "01" => oddParity <= '0';
```

```
            when "10" => oddParity <= '0';
            when "11" => oddParity <= '0';
        end case;
    end process;
end architecture b;
```

In VHDL all choices have to be matched in parallel (cf. [8], 3.2.2). This behavior can be achieved by synthesizing a n-muxer for each bit of each signal that is assigned in all code paths below the initial case expression, where n equals the number of bits of the case expression.

This transformation is very expensive as a case expression with just 8-bits leads to a synthesis of an 8-bit muxer for each assigned bit wide signal. A n-bit muxer can be constructed using $2^n - 1$ 1-bit multiplexers. An informal proof for this shall be given below:

Let mux be a function defined as

$$\text{mux}(\text{selector}, \text{sig}_0, \text{sig}_1) = \begin{cases} \text{sig}_0 & \text{if selector} = 0 \\ \text{sig}_1 & \text{if selector} = 1 \end{cases} \tag{3.1}$$

The function mux can also be defined using boolean logic. Equation 3.2 shows one possibility and table 3.1 defines both functions in terms of a truth table.

$$\text{mux}_{bool}(\text{selector}, \text{sig}_0, \text{sig}_1) = (\text{sig}_0 \wedge \overline{\text{selector}}) \vee (\text{sig}_1 \wedge \text{selector})) \tag{3.2}$$

Now, a n-ary muxer function can be defined recursively using equation 3.3. A 3-muxer for example, must be able to cope with 2^3 different input ports and can be created by using $k = 2$. The 3-bit selector is thus sufficient to select any of the given signals. Each recursion step, $(k-1)$, produces one muxer and binds the outputs of further two multiplexers onto its input. For mux_k there are $k+1$ steps in the recursion. Each step adds twice as much multiplexers to the circuit as the step before, accounting for an overall count of $2^{k+1} - 1$ multiplexers for mux_k[4].

$$\begin{aligned} mux_0(a,b,c) &= mux(a,b,c) \\ mux_k(s_k, iZ_{2^k-1}, iZ_{2^k-2}, \ldots, iZ_0) &= mux(s_{k-1}, \\ &\quad mux_{k-1}(s_{k-1}, iZ_{2^k-1}, iZ_{2^k-2}, \ldots, iZ_{2^{k-1}}), \\ &\quad mux_{k-1}(s_{k-1}, iZ_{2^{k-1}-1}, \ldots, iZ_0)) \end{aligned} \tag{3.3}$$

4 Figure 3.12 shows a netlist for a 3-bit multiplexer

Table 3.1 Truthtable for the equations 3.1 and 3.2

sig_0	$selector$	sig_1	$mux(selector, sig_0, sig_1)$
False	False	False	False
False	False	True	False
False	True	False	False
False	True	True	True
True	False	False	True
True	False	True	True
True	True	False	False
True	True	True	True

Semantics of if statements

In normal programming languages, if statements are a way to model branches in the execution path of the program. Depending on the evaluation of a condition, either the *true* path or the *false* path will be chosen. In VHDL's synthesis semantics, however, if statements are used to describe memories. Given the example code 3.33,

Listing 3.33 Simple conditional assignment

```
if (clock = '1' and clock'event) then
    A <= B;
end if;
```

A <= B appears as though it's only executed if the condition is true. This is simply not the case here. A connection between node A and B in the netlist is made either way, but it's the type of connection that matters. Instead of a simple wire going from B to A, a memory element with a certain conditional *runtime* behavior has to be utilized in between in order to achieve the modeled sequential behavior (cf. [7], 6.1).

As the figure 3.8 shows very clearly, an edge sensitive flip-flop is the synthesis result of listing 3.37's synthesis. This is the consequence of the controlling `if` condition at the top of listing 3.37. The semantics here are that A will only get assigned the current value of B, if and only if a rising edge on clock is detected.

Level sensitive storage elements like, for instance, D-Latches, can also be synthesized. Consider for example the snippet 3.34.

Listing 3.34 Simple D-Latch being utilized

```
if (foo + 3 < bar - 15) then
    A <= B;
end if;
```

Neither foo nor bar are declared to be clock signals but are just ordinary wires transmitting data. However, the assignment (in hardware) shall only be apparent on A if and only if the condition evaluates to true. So how does one get this behaviour in a unchangeable circuit. Like above, it comes down to discrete elements that can remember data (cf. [7], 6.2). In the synthesis result of code 3.34, the assignment is not done through a DFF. Instead a D-Latch sits between the signals A and B. As clock input for the latch serves the netlist for the if expression foo + 3 < bar − 15 (cf. section 3.3.4).

The only difference between listings 3.34 and 3.33 is the missing clock edge specification in listing 3.34. The following paragraph will inform about the nature of clock edges and explains the term synchronous condition.

Clock edge specification According to [7] 6.1.2 there are 10 ways how a clock edge can be described in VHDL. These are shown in code 3.35.

Listing 3.35 Clock edge specification syntax in VHDL

```
-- rising clock edge modelling
clock = '1'       and clock'event
clock = '1'       and not clock'stable
clock'event       and clock = '1'
not clock'stable  and clock = '1'
rising_edge(clock)
```

```
-- falling clock edge modelling
clock = '0'       and clock'event
clock = '0'       and not clock'stable
clock'event       and clock = '0'
not clock'stable and clock = '0'
falling_edge(clock)
```

falling_edge(clock) and rising_edge(clock), in particular, are more than just procedure calls abbreviating the above mentioned long versions of clock edge specificators. This is a detail of VHDL and not relevant here.

A Synchronous condition is a condition (i.e. expression) that contains a clock edge specification and is true only if the clock edge specification evaluates true. This connection can be made formal by the following boolean predicate 3.4.

$$\begin{aligned} \text{syncCond}(c) = \text{typeOfBool}(c) \wedge \\ \text{containsClockEdge}(c) \\ (c = \text{true} \rightarrow \text{clockEdge}(c) = \text{true}) \end{aligned} \tag{3.4}$$

As an example consider the VHDL expression:

```
en = '1' and (clock = '1' and clock'event)
```

Now, the according boolean formula for c can be extracted and used in the previously defined predicate.

$$\begin{aligned} \text{let } c := (\text{en} = \text{true} \wedge \text{clock}) \implies \text{clockEdge}(c) = \text{clock} \\ \text{syncCond}(c) = \text{typeOfBool}(c) \\ \wedge \text{containsClockEdge}(c) \\ \wedge ((\text{en} = \text{true} \wedge \text{clock}) = \text{true} \rightarrow \text{clock} = 1) \end{aligned} \tag{3.5}$$

Hence, a synchronized condition must be of type boolean, it must contain at least one clock edge and finally, the expression

$$((\text{en} = \text{true} \wedge \text{clock}) = \text{true} \rightarrow \text{clock} = 1)$$

must evaluate true for each possible binding of *en* and *clock*. In this case there are exactly 2^2 possible bindings.

3.3.4 Transformation algorithm – Synthesis examples

This chapter presents an own synthesis algorithm. Because of Yodl being the first open-source synthesis tool for VHDL – at least at the time of this writing – this was necessary, because every other available tool is closed-source and the IEEE standard [7] does not specify how the synthesis should be done, but rather explains *what* should be synthesizeable and what hardware representation shall be used for synthesis.

The transformation component, which ultimately generates a RTLIL netlist from a VHDL AST, is named NetlistGenerator. This synthesis component lies entirely in one class providing the public API summarized in listing 3.36.

Listing 3.36 Public API of NetlistGenerator

```
class NetlistGenerator {
public:
    NetlistGenerator(Yosys::RTLIL::Module *r) : result(r) {};
    int operator()(Entity *);

    Yosys::RTLIL::Module *result;
private:
    /* intentionally left out */
};
```

Code 3.36 shows the signature of the overloaded call operator. If a netlist generator object gets called with a valid pointer to an entity object, the entire entity will be traversed and the generated netlist will be located in result. The rest of this chapter shortly explains how the synthesis is done in particular. Note that every netlist presented here is the unmodified result of Yodl itself.

Synthesis of Entity objects

An `entity` object holds a list of port declarations. Those are used in order to describe what inputs or outputs the architecture can manipulate. Because of Yodl's reuse of vhdlpp's codebase, it is restrained to only one architecture for each given entity (cf. file `entity_elaborate.cc`, line 50). For that reason the netlist generator component does not need to take care about more than one architecture.

The said entity ports are synthesized to RTLIL wire objects (cf. section 3.3.1). Currently only ports of type std_logic or std_logic_vector can be synthesized. Every other type raises an error message.

Synthesis of Block and Process statements

In the current prototype, declarations in block or process statements won't be taken care of. The synthesizer only traverses the concurrent/sequential statements inside a block or process.

Synthesis of sequential statements

Section 3.3.2 introduces SVHDL which basically represents a strongly simplified format for VHDL AST's. According to this (informal) specification (see grammar in section 3.3.2), a sequential statement can only be a wait statement, a simple signal assignment statement a simple variable assignment statement or a case statement. This work is only concerned with simple signal assignment statements, case statements and if statements. Wait statements are not mandatory, as they mostly serve for purposes regarding checks for semantic correctness of the source program. Hence, they won't be considered in synthesis.

The following part of this section will show a few motivating examples of what the current synthesizer is capable of. Every example will show a listing, a netlists produced by the synthesis algorithm for the source and a corresponding explanation.

A simple synchronized assignment

Listing 3.37 Code for a synchronized bit assignment

```
-- libraries and entity decl deliberately omitted
-- A and clock are both of type std_logic
architecture behv of adder is
   function rising_edge(c : in std_logic) return std_logic;
begin
   process(A) is
   begin
      if rising_edge(clock) then
         A <= '0';
      end if;
   end process;
end behv;
```

As can be seen in the `process` part of listing 3.37, the signal A shall only be driven if and only if a rising edge in signal clock occurs. If there is no rising edge, the current value needs to be stored. This can be achieved through the usage of a simple edge sensitive flip-flop[5] also known as D-flip-flop (DFF for short). Yodl transforms listing 3.37 into the netlist shown in figure 3.8. The netlist also shows, how the signals are connected to represent the semantics from the listing. The values in the elliptical shapes represent constant values, whereas strings contained in diamond shapes are used to denote driving signals. The netlist format identifies all driven signals with an octagonal border. Moreover, the cells (aka. functional building blocks) are easy to spot, because of their $-naming scheme. A muxer cell, for example, is always labeled by the string *$-mux*. "BUF" nodes are not relevant, because they don't add any logic to the netlist and thus shall be neglected.

Nested synchronized assignment

Listing 3.38 Code for a nested synchronized bit assignment

```
-- [...]
-- A and clock are both of type std_logic
architecture behv of adder is
   function rising_edge(c : in std_logic) return std_logic;
begin
   process(A) is
   begin
      if rising_edge(clock) then
         if rising_edge(clock) then
            A <= '0';
         end if;
      end if;
   end process;
end behv;
```

Listing 3.38 contains an assignment that gets doubly synchronized by the two enclosing if statements. Commercial tools would probably report a warning or, in some contexts, an error. Yodl, however, does not complain as standard [7] does not explicitly forbids this kind of synthesis behavior (cf. chapter 6.1). The netlist in figure 3.9 clearly shows the result of such a nesting. As opposed to circuit 3.8, the nested version needs one additional clock cycle for '0' to appear on output A.

5 See chapter 11.4 in [9]

Simple latched assignment

Listing 3.39 Code for a simple latched bit assignment

```
-- same libraries as above
-- A, B and C are of type std_logic
architecture behv of adder is
   function rising_edge(c : in std_logic) return boolean;
begin
   process(A) is
   begin
      if A = B then
         C <= '1';
      end if;
   end process;
end behv;
```

Level sensitive flip-flops are commonly called *latch*. DFF's only sample values at clock edge, whereas latches maintain a constant connection between their inputs and outputs if the signal value is either zero or one (cf. [9], 11.3). For instance, suppose a latch is low-level active. This latch would only interconnect its input and output only if the enable input equals zero. DFF's also possess input pins like *enable*, but in this case those inputs are commonly called *clock*, because of the emphasis on clock edge synchronicity.

Listing 3.39 contains a single if statement with an ordinary condition on top of it. It does not have an else-path. As a consequence, a register-like hardware cell must be used to handle the case when the condition A = B does not apply. If it does, C will be driven with the constant value of 1. Otherwise, the previous value (in this case always 1) will be stored. Note how 3.10 connects the netlist for the condition directly with the *EN* input of the latch.

In digital circuits, particularly in synchronous circuits, the usage of latches is mostly unwanted as latches can't generally be used for feedback assignments like: A <= A + 1; However, this is a topic far beyond the scope of this work. References for further reading are 3.6.2, 3.6.1 and, most importantly 3.4 from [8].

Nested, latched assignment

Listing 3.40 Code for a nested latched bit assignment

```
-- [...]
-- same preamble as above
   process(A) is
   begin
      if A = B then
         if not A then
             C <= '1';
         end if;
      end if;
   end process;
end behv;
```

Analogous to the snippet in listing 3.38, latches can be cascaded too. The relationship between the netlists 3.11 and 3.10 exactly correspond to 3.8 and 3.9. Consequently, a description of if would be redundant.

Figure 3.8 Netlist for listing 3.37

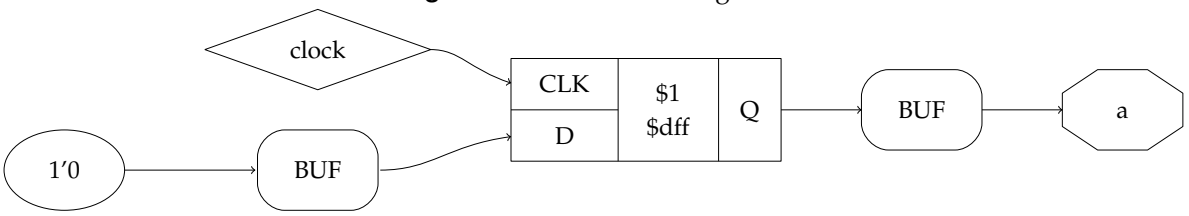


Figure 3.9 Netlist for the listing 3.38

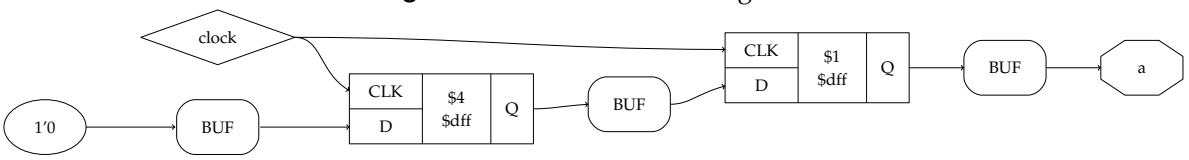


Figure 3.10 Netlist for listing 3.39

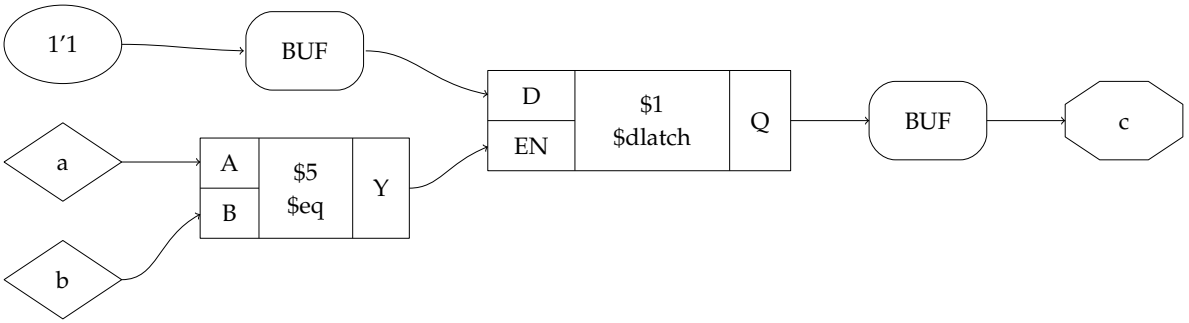
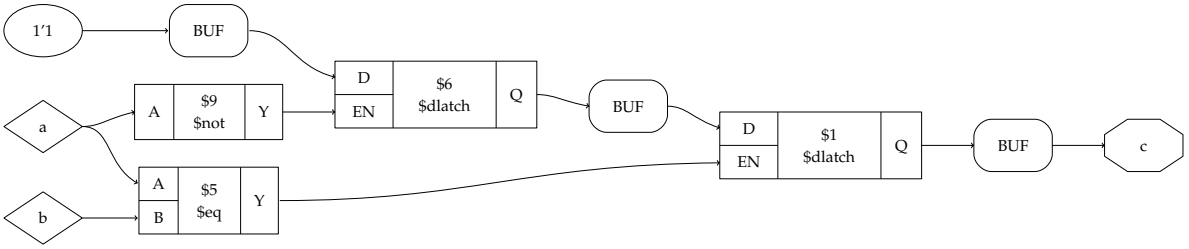


Figure 3.11 Netlist for listing 3.40



Simple case statement

Listing 3.41 Code for a simple case statement

```
-- same libraries as before
-- A is a std_logic and baz a std_logic_vector(2 downto 0)
architecture behv of caseT is
begin
   process(A) is
   begin
      case baz is
         when "000" => A <= '0';
         when "001" => A <= '1';
         when "010" => A <= '1';
         when "011" => A <= '1';
         when "100" => A <= '0';
         when "101" => A <= '1';
         when "110" => A <= '1';
         when "111" => A <= '1';
      end case;
   end process;
end behv;
```

VHDL's case statement's are particularly interesting for synthesis because all test from all choices must be performed parallel in hardware. Listing 3.41 shows a simple example of a case statement being used to model a 3-muxer. The fact that, for instance, A <= '0', could be replaced by an arbitrary sequence of sequential statements shall be neglected for now.

The code from 3.41 can be interpreted as: "if baz equals the signal vector containing 000, A shall be driven by '0' ...". The netlist 3.12 presents an implementation of this behaviour. Note that $-mux cells are equivalent to the abstract multiplexers from section 3.3.3. Figure 3.12 shows that each muxer has three inputs and a single output. Analogous to the the muxer semantics from 3.3.3, A gets selected if S equals zero, otherwise B.

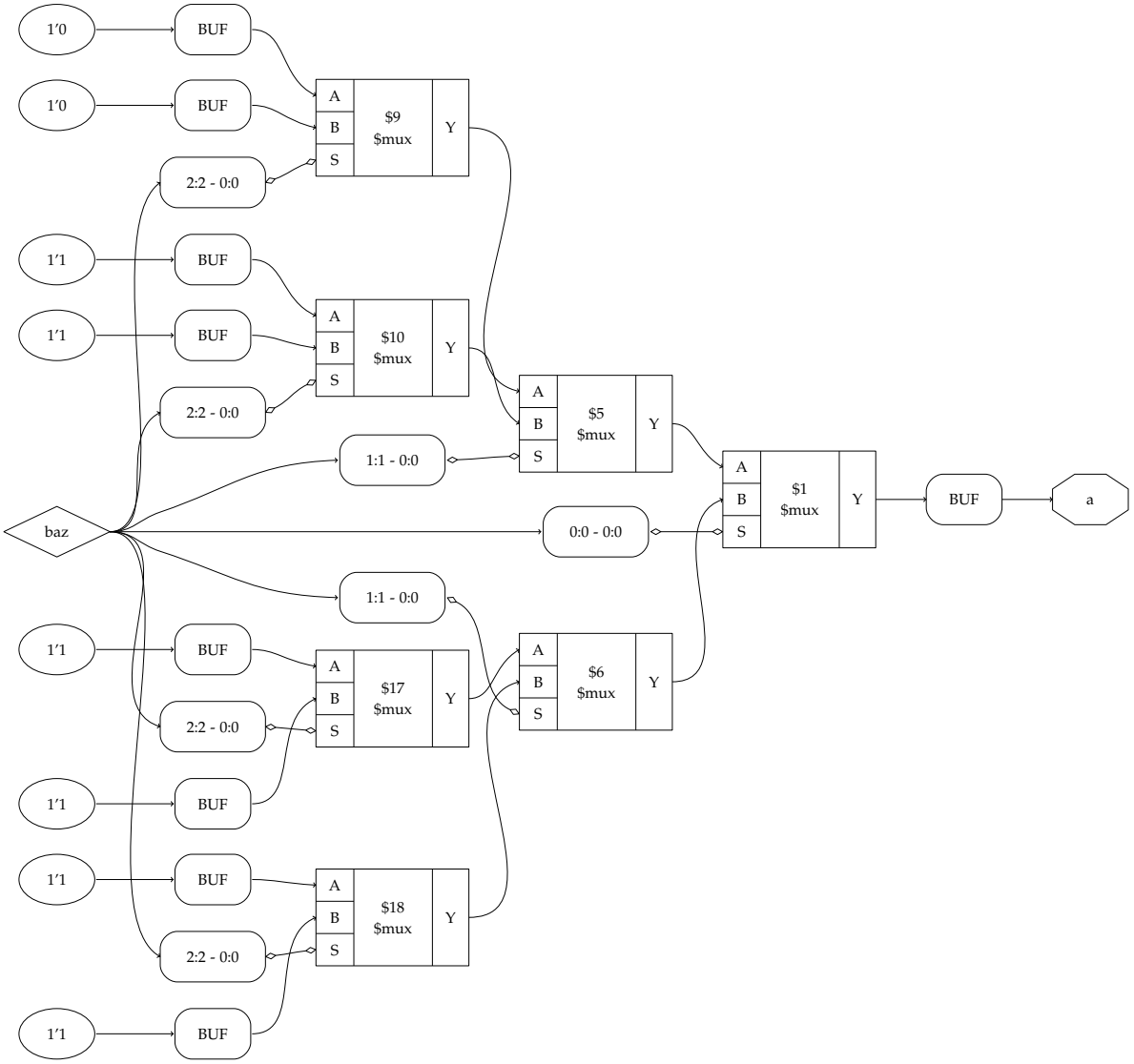
Note that nodes with rounded corners connecting the various muxer selectors with the signal baz show which bit is being connected by the respective edge. For example $2{:}2 - 0{:}0$ means that the slice 2:2 (just one bit) is connected to the zeroth bit on the other side.

Nested case statements

Listing 3.42 Code for three nested case statements

```
-- same libraries as before
-- A, B, C and sum are ports of type std_logic
architecture behv of adder is
begin
   process(A) is
   begin
      case A is
         when '0' => case B is
                        when '0' => sum <= '0';
                        when '1' => sum <= '1';
                     end case;
         when '1' => case C is
                        when '0' => sum <= '0';
```

Figure 3.12 Netlist for listing 3.41



```
                        when '1' => sum <= '1';
                    end case;
        end case;
    end process;
end behv;
```

Case statements, in VHDL, can be arbitrarily deep nested as example code 3.42 shows. As can seen from the snippet, if signal A equals '1' and C equals '0', sum will be driven by the value '1'. The three other paths follow analogous. Of course, the same functionality couldn't be achieved through the usage of a single case statement whose condition expression is composed of a 2-bit signal, because both inner case statements use two different input signals as selectors.

In 3.42 it appears as if the case statements are going to be evaluated from outside to inside. Concretely, it seems as though first signal A is matched to either '0' or '1' and according to the result, the interpreter evaluates either the first or the second path. In hardware, however, there is no notion of evaluation, because the circuit is static and cannot change during its runtime. For better illustration of how a synthesis algorithm works, nested control statements must be read from the inside to the outside. Following this guideline, figure 3.13 is easier to comprehend with. Given the first path of 3.42

```
case A is
    when '0' => case B is
                    when '0' => sum <= '0';
```

the constant value '0' connects to the A input port of the muxer for the case statement containing B as conditional expression. Because of its hardware representation as muxer, the inner case statement itself possesses an output bit for every element of the disjoint set of driven signals below the root of the case statement. Therefore, this output signal serves as input to the outer case statement if and only if A equals '0'.

Synchronized simple case statement

Listing 3.43 Code for a simple synchronized case statement

```
-- [...]
-- A, clock are ports of type std_logic and
-- sel is a std_logic_vector(2 downto 0)

architecture behv of syncCase is
    function rising_edge(c : in std_logic) return std_logic;
begin
    process(A) is
    begin
        if rising_edge(clock) then
            case sel is
                when "000" => A <= '0';
                when "001" => A <= '1';
                when "010" => A <= '1';
                when "011" => A <= '1';
                when "100" => A <= '0';
                when "101" => A <= '1';
                when "110" => A <= '1';
                when "111" => A <= '1';
```

```
        end case;
      end if;
   end process;
end behv;
```

Listing 3.43 illustrates how assignments occurring inside the choice statement lists of case structures can be synchronized. As can be seen in the code, the choices are exhaustive, which means that every possible combination of the bits in sel occurs only once as option for a possible code path. VHDL-2008 does require semantic checks in this regard, but Yodl currently does no such thing, since a proper validity check suite would far exceed the scope of this work (cf. [6], 10.9).

As can be seen in the comment on top of 3.43, sel is a signal vector with a width of three bits. Hence, a 3-muxer is shown in the rendered netlist 3.14 belonging to the code. The netlist for the muxer depicted in 3.14 is exactly equivalent to that shown in 3.12. However, the output of this structure does not simply drive the signal A but rather feeds the data port (D) of the preceding DFF. Furthermore, figure 3.14 confusingly shows 8 equal connections between the 3-muxer's output and the data input of the DFF. This is a result of the inner workings of the synthesis algorithm and not relevant here. Yosys provides for a optimization pass that eliminates duplicated (and equal) connections anyway and, as a consequence, the algorithm does not need to be adjusted with regard to this issue.

If statement representing a muxer

Listing 3.44 Code for an if statement actually representing a case statement (aka. muxer)

```
-- [...]
-- A, B and C are ports of type std_logic
architecture behv of adder is
   function rising_edge(c : in std_logic) return boolean;
begin
   process(A) is
   begin
      if A = B then
         C <= '0';
      else
         C <= '1';
      end if;
   end process;
end behv;
```

The last code excerpt 3.44 shows that, given the right preconditions, an if statement does not produce synchronized netlists whatsoever. One could argue that in this case, the if statement is nothing else than a case statement. In the current context he/she would be completely right. Yodl's synthesis algorithm actually converts 3.44 into a semantically equivalent case statement AST before it continues its synthesis.

For that reason the code in 3.44 leads to the same netlist as the source code 3.45.

Listing 3.45 Equal semantics as in 3.44

```
-- [...]
case A = B is
   when TRUE  => C <= '0';
   when FALSE => C <= '1';
```

Figure 3.13 Netlist for listing 3.42

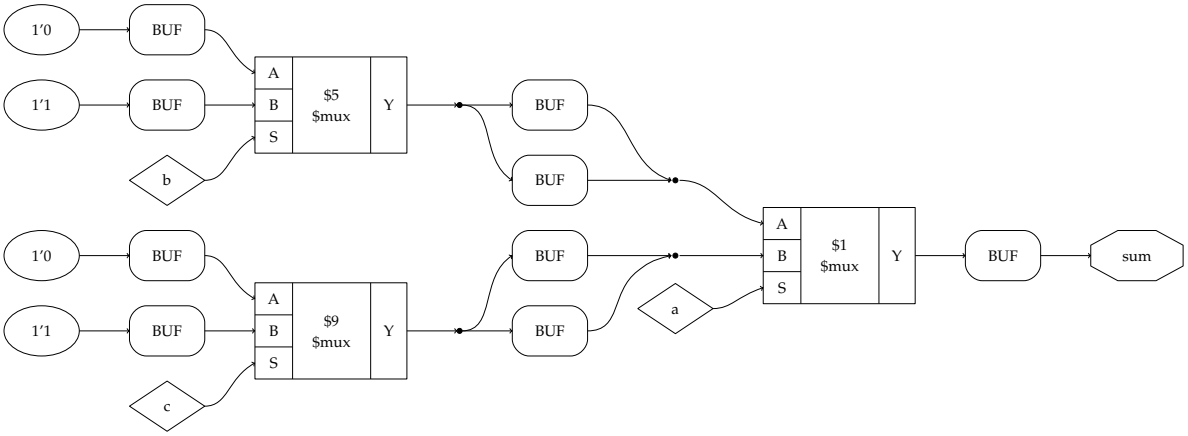


Figure 3.14 Netlist for listing 3.43

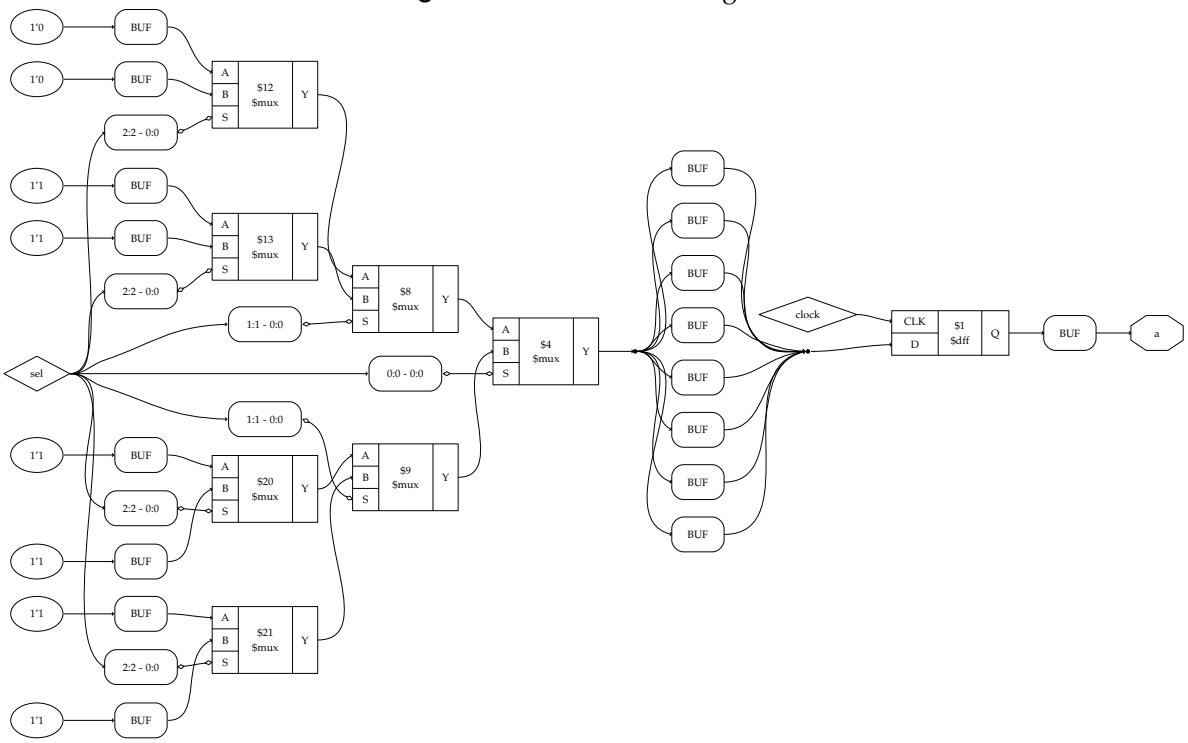
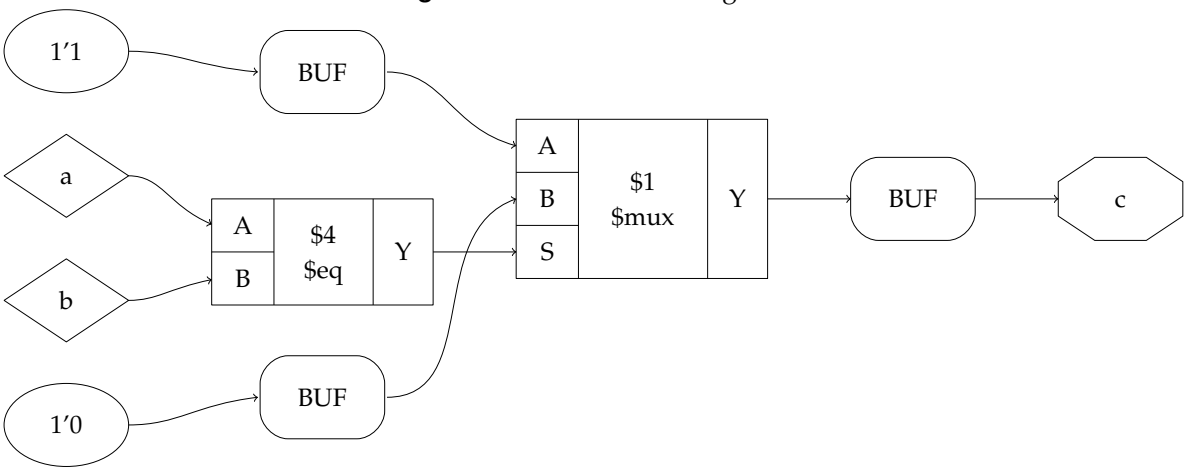


Figure 3.15 Netlist for listing 3.44




```
end case;
-- [...]
```

3.3.5 Transformation algorithm – Implementation details

During traversal, every time the algorithm encounters an if or case statement, it first examines the conditions and bodies of those structures. Based on this information, the synthesizer pushes a netlist object on top of a special stack which contains references to all input and output ports as well as some additional information like, for example, the corresponding control structure resulting in the netlist.

Introduced abstractions

The netlist generator object carries around a stack containing elements of type stack_element_t. As can be seen in file `netlist_generator.h`, the stack is represented by a STL vector (cf. Glossary). The algorithm only adds new elements using push_back and only removes them with pop_back. However, the container must be a vector, because the algorithm needs to be able to randomly access the structure when synthesizing signal assignments.

Listing 3.46 API of base stack element

```
class NetlistGenerator {
public:
    // [...]
    struct stack_element_t {
        // constructors intentionally omitted
        std::map<string, netlist_element_t *> netlist;
        std::set<string> occurringSignals;
    };
    // [...]
};
```

Objects of type struct_element_t possess the API depicted in listing 3.46. There are three classes that inherit from struct_element_t. As code 3.46 illustrates, every element on the stack

carries an association between signal IDs (strings) and netlist_element_t objects as well as a list of all driven Signals. These two members describe the core of the current context. For understandability purposes, code excerpt 3.47 shows one of the derived classes used for the case statement context. If the algorithm, for example, would encounter a case statement like the one in 3.43, it'd first search all possible traversal paths below the subjected case statement for driven signals creating a set of those signals along the way. In the context of 3.43 this set would only contain one element: A. After that, the synthesizer would construct a appropriate netlist for each of the set's elements. Finally, the set, and the association of the signals with their muxer netlists is packaged into an object of type case_t and pushed to the stack (cf. file `netlist_generator.cc`, function executeIfStmt in line 1041 and 1096, and function executeCaseStmt).

Listing 3.47 Case statement context class derived from stack_element_t

```
class NetlistGenerator {
public:
    // [...]
    struct case_t : stack_element_t {
        // constructors intentionally omitted
        Yosys::RTLIL::SigSpec curWhenAlternative;
    };
    // [...]
};
```

The synthesis algorithm does the same for if statements. Regarding this, however, there are a few subtleties that need further attention. As the synthesis results for 3.37, 3.39 and 3.44 show, netlists for if statements depend on two factors: The if condition and the set of disjoint driven signals below the root of the respective if block. Depending on the kind of condition of the if expression and the distribution of the various statements inside the if block, the contained signal assignments will be latched, clock accurately synchronized or only carried through a simple multiplexer. Consequently, there are two other classes implementing the common interface presented in 3.46: Class if_dff_t and if_latch_t; both of which being declared by the `struct` keyword making every member public by default as opposed to the keyword `class`. The two classes, however, don't add anything new to the base interface from 3.46 but only serve as a kind of enumeration on the type level. This is a common design pattern in functional programming and becomes feasible in C++ through the use of the pattern matching library Mach7.

An open point of this elaboration still remains: Objects of type netlist_element_t. Like stack_element_t, a netlist element constructs an interface using an abstract class. As 3.48 shows, this API solely consists of one member named output. While some netlists (e.g. Multiplexers or RAM blocks) can have inputs with attributes associated with them, others only have one ore more *anonymous* inputs. For example, inputs of multiplexers have to carry the selection semantics with them, because the synthesis algorithm needs to know which muxer input corresponds to which choice in the corresponding case statement. On the other hand, netlist parts, such as latches or DFFs, only ever have one possible input.

Listing 3.48 API of base netlist element

```
class NetlistGenerator {
public:
    // [...]
    struct netlist_element_t {
```

```
        // constructors intentionally omitted
        Yosys::RTLIL::SigSpec output;
    };
    // [...]
};
```

3.4 Current Limitations

The following is a non-comprehensive list of the most important limitations of Yodl.

- Only one `architecture` definition per `entity` declaration is allowed.
- Configurations are not supported, because the underlying parser does not support this yet (cf. file `parse.y` in line 650).
- Yodl can't fully parse VHDL-2008. The code snippet below shows that, for example, a list of named arguments can't be used as a parameter list for a function called inside of an expression context (cf. `parse.y`).

```
architecture beh of ent is
   -- some decls
begin
   -- this should be parsable, but produces syntax error
   result <= foo(fnord => 3, foobar => 4)(3);

   -- this is parsable
    result <= foo(4, 3)(3);
end beh;
```

- The synthesis algorithm is in a proof-of-concept pre-alpha state. It is untested and does not support signal or variable assignments with a vector subscription as left-hand side expression.
- The synthesis implementation can not translate variable assignments.
- The implementation is unable to cope with signal assignments like A <= A + 1, i.e. assignments where the left-hand side also occurs, possibly multiple times, in the right-hand side expression.
- Synthesis of complex types (e.g. arrays of arrays of records ...) is not possible yet.
- The test suite is still rudimentary.
- The library and package concept was not considered in this work.
- The complex visibility rules from chapter 12 of [6] were not considered, since Yodl's base, vhdlpp, already came with an implementation of the scoping and visibility rules.
- Component instantiations can't be processed.

4 Yodl – Future work

4.1 Complete parser

Section 3.4 has already made clear, that the current parser solution is unfinished and hence unable to parse the entirety of VHDL-2008. Thus, a very important future goal of Yodl is to completely refactor or rewrite the parser component. This is likely to be difficult because of the reasons denoted in section 2.1

The following section describes a tool that offers a lot of help if the parser actually needs to be rewritten.

4.1.1 BNFC and LBNF

LBNF is a formalism which is based on the notation system BNF. The L in LBNF stands for *labeled*. Like BNF, the LBNF notation is used to describe context free grammars, but unlike BNF it forces the writer of the grammar to annotate every rule with a certain label.

For example, a simple expression grammar would be given in LBNF using

```
ENum . Exp3 ::= Integer ;
EMul . Exp2 ::= Exp2 "*" Exp3 ;
EPlu . Exp  ::= Exp "+" Exp2 ;
_    . Exp  ::= Exp2 ;
_    . Exp2 ::= Exp3 ;
_    . Exp3 ::= "(" Exp ")" ;
```

BNFC is a program generator that takes LBNF code as input and produces a complete frontend for the specified language, given the fact that the grammar is sound. It can output language frontends in different languages. At the time of this writing, BNFC is able to generate Haskell, OCaml, C/C++, C-Sharp and Java code. For the previously defined expression grammar, BNFC would output a flex and bison file describing the scanner and parser part and a data model for the abstract syntax tree. This data model in turn is used in the bison file in order to actually create the said AST. The according C++ classes for the grammar roughly look like listing 4.1.

Listing 4.1 Generated classes for expression grammar

```
class Exp { public: virtual ~Exp() = default; };
class ENum : Exp { public: int value; ENum(int v) : value(v) {}  }
class EPlu : Exp {
    public: Exp *l, *r;
    EPlu(Exp *le, Exp *ri) : l(le), r(ri) { }
};
class EMul : Exp {
    public: Exp *l, *r;
    EMul(Exp *le, Exp *ri) : l(le), r(ri) { }
};
```

Normally the above classes need to be handcrafted. This was done for Yodl' parser. For anything but trivial grammars, this task is tedious and error prone and should be automated. Refer to [22] for details about the labeled BNF formalism.

In the scope of this work, the entire VHDL-2008 BNF grammar has already been extracted from the official standard and completely *translated* into LBNF (cf. `vhdlpp_parser/-newparser/vhdl-2008/vhdl-2008-all.lbnf`).

4.2 Further grammar issues

Yodl's current parser demonstrates how especially reduce-reduce ambiguities can be dealt with. Simply put, the parser needs to carry around a stack of scopes that it currently processes (cf. `parse.y`, in particular line 366 pushes a new scope frame and line 385 pops the same). Since every RR conflict arises because of VHDL's use of parentheses for array subscriptions, function/procedure calls and type declarations (see listing 4.2), the parser must know about the scope it currently parses and every already declared/defined symbol that scope possesses.

Listing 4.2 Illustration of a common reduce-reduce conflict

```
function foo () is
begin
-- [...]
return "0000";
end function foo;

signal foo : std_logic_vector(7 downto 0);

foo(0); -- This could be parsed as
        -- 1. subscription of the vector foo
        -- 2. as call to the 0-ary function foo with one argument,
        --    which would be syntactically correct but semantical
           non-sense
        -- 3. as subscription of the return value resulting from the
        --    call to the 0-ary function foo with no argument
```

Hence, there is a solution for dealing with RR conflicts, but what is about shift-reduce conflicts? They aren't allowed to lift the grammar into the set of non-deterministically context free languages because that would mean that even with Bison's GLR feature enabled the grammar couldn't practically be processed. A parse-forest would result from a parser run, which is completely inacceptable for production-aspiring compilation systems.

For that reason, it must be proven that all SR conflicts together (without the RR) indeed don't make the grammar non-determinstic. This work does not provide any proof of that kind, because of the complexity of the problem. The next paragraph, however, shows a first approach.

GLR is a parser algorithm which, simply put, duplicates itself if it encounters an ambiguity. Each parser then continues virtually in parallel. Each duplicate can of course in turn duplicate and split itself again and again if ambiguities are hit very often. *If it is guaranteed, that for each split only one of the duplicated parsers succeeds, the parser produces only one parse tree for all inputs over the grammar.* This condition is the core of the previously mentioned proof!

4.3 Complete VHDL support

Currently Yodl is mainly an experimental project that does not support VHDL specific language concepts such as packages, libraries, components, configurations, generics and multiple architectures for a given entity. As every industry-quality synthesis tool supports those features, Yodl will support them too in the near future.

Packages and libraries are not trivial to implement because of the complicated visibility rules described in chapter 12 of [6] Especially generics provide for another challenge as 6.5.6.2 of [6] allows for generic functions and types which renders them more like generics found in the programming language ADA as opposed to the older concept where they were handled as simple constants.

4.4 Far in the future

The two last sections elaborate two projects that are probably very work intensive. Hence, they won't be concerned until a first prototype for full VHDL-2008 is released.

4.4.1 Formal specification of VHDL's synthesis semantics

The Book [10] already provides formal semantics for VHDL. Since its initial release, which was in 1995, a lot of VHDL's synthesis semantics has changed however. Because of that, a new specification becomes necessary.

4.4.2 Regression based test suite

As section 3.1.7 already states, the automation of tests is important to maintain a correct code base. Compilers are among the most complex and complicated software systems in existence which makes formal verification very hard. Smoke tests, as they are implemented in the scope of this work, are not quite sufficient because of their inadequate code coverage. A Regression test suite in the sense of 3.1.7 solves the coverage issue.

Bibliography

[1] VHSIC Program Office, *Very High Speed Integrated Circuits - VHSIC - Final Program Report*, Defense Technical Information Center, 1990.

[2] Clifford Wolf, *The Yosys Reference Manual*

[3] Florian Mayer, *Das HLS-Framework LegUp – Aufbau, Funktionsweise und Anwendung*, FH Rosenheim, 2016

[4] Maribel Fernandez, *Programming Languages and Operational Semantics*, Springer, London, 1st edition, 2014.

[5] David A. Schmidt,
Denotational Semantics: A Methodology for language development,
1997, Kansas State University,
https://www.scss.tcd.ie/Andrew.Butterfield/
Teaching/CS4003/DenSem-full-book.pdf

[6] IEEE Computer Society,
IEEE Standard VHDL Language Reference Manual,
IEEE Std 1076-2008

[7] IEEE Computer Society,
IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis,
IEEE Std 1076.6-2004

[8] Jürgen Reichardt, Bernd Schwarz,
VHDL-Synthese
6. Auflage, 2013, Oldenbourg Verlag

[9] Jürgen Reichardt,
Lehrbuch Digitaltechnik,
3. Auflage, 2013, Oldenbourg Verlag

[10] Carlos Delgado Kloos, Peter T. Breuer, *Formal Semantics for VHDL*, 1995, ISBN-13: 978-1461359418

[11] Maciej Sumiński, *Adding VHDL support to Icarus Verilog*, FOSDEM, Brussels, 01.02.2015

[12] http://www.graphviz.org/, *Graphviz - Graph Visualization*

[13] Stephen Williams,
VHDLPP source code,
`https://github.com/steveicarus/iverilog/tree/master/vhdlpp`

[14] Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup,
Open Pattern Matching for C++, 2013

[15] John R. Bandela,
`https://github.com/jbandela/simple_match`,
`https://github.com/CppCon/CppCon2015/tree/master/Presentations`,
`CppCon talk at https://www.youtube.com/watch?v=9IVCVSwn-fI`,
Simple, Extensible Pattern Matching in C++-14, CppCon 2015

[16] Miran Lipovaca,
Learn You a Haskell for Great Good!: A Beginner's Guide,
April 2011, ISBN-13: 978-1593272838

[17] The SQLite Consortium,
SQLite testing,
`https://www.sqlite.org/testing.html`

[18] Clifford Wolf,
Vlog Hammer,
`http://www.clifford.at/yosys/vloghammer.html`

[19] *Catch – A modern, C++-native, header-only framework for unit-tests*,
`https://github.com/philsquared/Catch`

[20] *CppUTest unit testing and mocking framework for C/C++*,
`http://cpputest.github.io/`

[21] *On the Translation of Languages from Left to Right*,
Donald E. Knuth, 1965

[22] Markus Forsberg, Aarne Ranta,
The Labelled BNF Grammar Formalism,
`http://bnfc.digitalgrammars.com/LBNF-report.pdf`,
Department of Computing Science,
Chalmers University of Technology,
University of Gothenburg, 2005