University of Applied Sciences Western Switzerland
MSE - Software Engineering

Deepening project

KlugHDL : a SpinalHDL diagram generator

Author:
Julmy Sylvain

Supervisor:
Mudry Pierre-André

Fribourg, January 16, 2017

Contents

1 Introduction **3**
- 1.1 Context . . . 3
- 1.2 Goal . . . 4
- 1.3 Document overview . . . 4

2 Analysis **5**
- 2.1 What do we want ? . . . 5
- 2.2 SpinalHDL . . . 6
  - 2.2.1 Component . . . 6
  - 2.2.2 Abstract Syntax Tree . . . 6

3 Diagrams modelisation **8**
- 3.1 Hierarchy visualization . . . 8
  - 3.1.1 Hierarchical layout . . . 8
  - 3.1.2 Tree view . . . 9
  - 3.1.3 Multiple diagrams . . . 10
- 3.2 A visual example . . . 11
- 3.3 Model representation . . . 11
- 3.4 Conclusion . . . 11

4 Viewing library **12**
- 4.1 GraphStream . . . 12
  - 4.1.1 Implementation of the base graph model . . . 12
  - 4.1.2 Remarks . . . 13
  - 4.1.3 Conclusion . . . 14
- 4.2 Draw2D . . . 15
  - 4.2.1 Implementation of the base graph model . . . 15
- 4.3 Graph layout . . . 17
- 4.4 Conclusion . . . 18

5 Parsing the AST **19**
- 5.1 Diagram parsing . . . 19
- 5.2 Components parsing . . . 20
- 5.3 Ports parsing . . . 21
- 5.4 Connections parsing . . . 22
  - 5.4.1 Input connections . . . 22
  - 5.4.2 Output connections . . . 23
  - 5.4.3 Parent connections . . . 23
- 5.5 Conclusion . . . 24

6 Diagram visualization **25**
- 6.1 Intermediate representation . . . 25
  - 6.1.1 The DOT backend . . . 25
  - 6.1.2 The JSON backend . . . 26
- 6.2 Static visualization . . . 26
- 6.3 Dynamic visualization . . . 26

6.3.1 Extending the draw2d library 27
6.3.2 Problems 27

7 Conclusion **29**

Appendices **31**

A Facts sheet : Generate a component diagram **32**

B Facts sheet : Display, hide and toggle the view of the edge's type **33**

C Facts sheet : Enter a component **34**

D Facts sheet : Exit a component **35**

E Facts sheet : View a component diagram **36**

F Facts sheet : Display, hide, toggle the hierarchical tree view **37**

G Facts sheet : Generate the RTL **38**

H Facts sheet : Filter the view **39**

1 Introduction

A program is a sequence of mathematical instruction which are written by a human and executed by a computer. The VHDL langage has been created to describe digital and signals systems such as field-programmable gate arrays and integrated circuits[1].The problem is that VHDL is an old, verbose and tricky language.

SpinalHDL has been created to fight thos disadvantage. The main goal of a DSL (Domain Specific Langage) is to offer a more plaisant way to code and that's what dose SpinalHDL. Has written before, the code is written by a human and by using SpinalHDL, the human became happier than using just the VHDL.

KlugHDL is an another tools to make the human happiest, it's a component diagram generator which offer some

1.1 Context

SpinalHDL is a programming language to describe digital hardware and generate the corresponding in VHDL (or Verilog). SpinalHDL is written in Scala as a DSL and has multiple advantages[2] :

- No restriction to the genericity of your hardware description by using Scala constructs
- No more endless wiring. Create and connect complex buses like AXI in one line.
- Evolving capabilities. Create your own buses definition and abstraction layer.
- Reduce code size by a high factor, especially for wiring. Allowing you to have a better visibility, more productivity and fewer headaches.
- Free and user friendly IDE. Thanks to scala world for auto-completion, error highlight, navigation shortcut and many others
- Extract information from your digital design and then generate files that contain information about some latency and addresses
- Bidirectional translation between any data type and bits. Useful to load a complex data structure from a CPU interface.
- Check for you that there is no combinational loop / latch
- Check that there is no unintentional cross clock domain

The code 1 shows a AND gate written with SpinalHDL with the corresponding generated VHDL code, we could see the similarity between the two code.

```scala
import spinal.core._

class AND extends Component
{
    val io = new Bundle
    {
        val a = in Bool
        val b = in Bool
        val c = out Bool
    }

    io.c := io.a & io.b
}
```

```vhdl
entity AND_1 is
    port(
        io_a : in std_logic;
        io_b : in std_logic;
        io_c : out std_logic
    );
end AND_1;

architecture arch of AND_1 is

begin
  io_c <= (io_a and io_b);
end arch;
```

Listing 1: Example of a AND gate written in SpinalHDL and the corresponding generated VHDL code

1.2 Goal

The goal of the project is to produce an application which is analysing a spinalhdl program in order to produce a block diagramm of the corresponding hardware description. This application should offer the following activities :

- Display the complete graph of the programm
- Navigate through all the hierarchical level of the programm (Component inside another component)
- The edges should display some information about the signals :
  - type
  - input or output
  - ...
- A Hierarchical view (like in filesystems or in Quartus)
- A filter functionnality in order to view only some specific component

The application is going to be develop, firstable, in a standalone version and next could be develop into a plugin version for Eclipse or IntelliJ.

1.3 Document overview

TODO

2 Analysis

In this chapter, we would discuss about the "What do we want ?" questions. It means what need the final product based on the Goal of the project. The other part is about the actual state of SpinalHDL. We are not presenting all of the implementation or conception, just a small part which is usefull for this project. FInally, we introduce the concept of Component and AST (Abstract Syntax Tree).

2.1 What do we want ?

Based on the goal of the project at section 1.2, we expose what the end user of SpinalHDL want to do with KlugHDL. The figure 2.1 show the use case of KlugHDL.

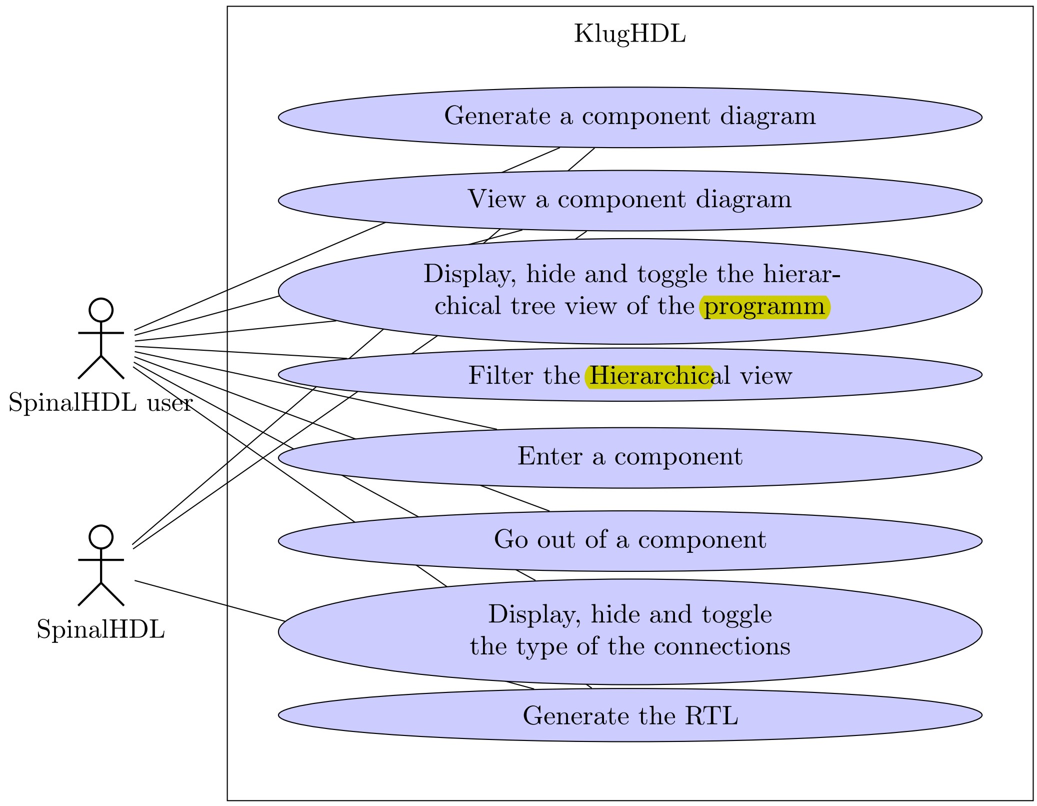

Figure 2.1: Use case of KlugHDL

An important point is the idea of port. If we want to design a HDL diagram, we have to display the types of the connection between the component. We have at least two alternative :

- display the types using the edges
- display the types using some ports attached to the component

We need to figure out which alternative is the best in order to display the type of a connection.

2.2 SpinalHDL

As explain in chapter 1, SpinalHDL is a DSL for VHDL programmation. SpinalHDL run with various internal construction such as an AST or a Component Tree. Those are the majors element needed for the diagram generation.

2.2.1 Component

A SpinalHDL's component is the base class of every SpinalHDL Program, like in the listing 1. Like in Verilog and VHDL, you can define components that could be used to build a design hierarchy. But unlike them, you don't need to bind them at instantiation [2]. That's why there is just one component against the architecture and entity combination in VHDL.

The component is the main p of the SpinalHDL architecture for this project because all the component are linked together with :

- a parent-children relationship
- a brother relationship

A component could contains and communicate with an other component, that's the parent-children relationship, and a component could communicate with the component contained by his parent, that's the brother relationship. This two relation could be seen in listing 2. The component is communication with their children which are communicationg between them.

A component could have other element inside him too, like Area or Function. But they are not relevant for the diagram generation.

2.2.2 Abstract Syntax Tree

An abstract syntax tree is a tree where the node are marked by operator and the leaf are marked by the operand of those operator. For example, the expression $5 * 2 + 3$ has his equivalent AST showed in the figure 2.2.

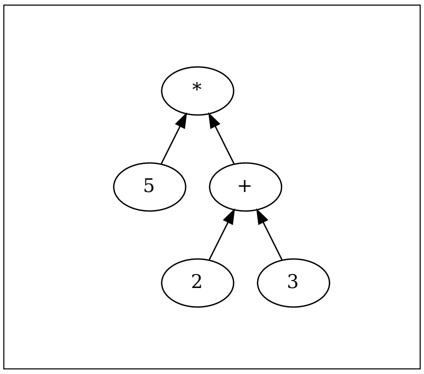

Figure 2.2: AST of the expression $5 * 2 + 3$

The AST is important because he is using to recover all the input and the output of a component, including the source of each input's component. To do that, we just need to recursively walk inside the AST of a SpinalHDL program like explain in the chapter 5.

```
class ParentChildrenBrother extends Component
{
    val andGate = new AndGate
    val xorGate = new XorGate
    val andXorGate1 = new AndXorGate
    val orGate = new OrGate
    val andXorGate2 = new AndXorGate

    val io = new Bundle
    {
        val a = in Bool
        val b = in Bool
        val c = in Bool
        val d = out Bool
        val e = out Bool
    }

    andGate.io.a := io.a            // children communication example
    andGate.io.b := io.b
    xorGate.io.a := io.b
    xorGate.io.b := io.c

    orGate.io.a := andGate.io.c     // brother communication example
    orGate.io.b := xorGate.io.c

    andXorGate1.io.a := io.a
    andXorGate1.io.b := io.b

    andXorGate2.io.a := io.a
    andXorGate2.io.b := io.b

    io.d := andGate.io.c & xorGate.io.c & orGate.io.c
    io.e := andXorGate1.io.c | andXorGate1.io.d | andXorGate2.io.c | andXorGate2.io.d
}
```

Listing 2: There is two different type of connections with SpinalHDL, a brother to brother ones and a parent-children ones

3 Diagrams modelisation

In order to produce a visual schema from a SpinalHDL program, we need to parse his AST and produce an intermediate model for the diagrams. The reason to use an intermediate model is that we could then build multiple backend generator for multiple output target. The complete pipeline of the generation is visible in figure 3.1.

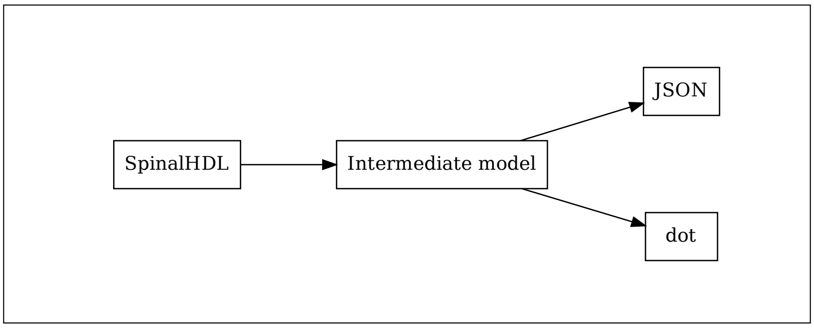

Figure 3.1: This figure show the entire generation pipeline use by KlugHDL in order to generate different output. The intermediate model is use to produce the target code, for example dot or JSON file

We also need to introduce an crucial element to understand how the model is build : the hierarchy visualization.

3.1 Hierarchy visualization

The diagrams produces by KlugHDL are kind of special, they are hierarchical ones. That's means we need to find a way to show the hierarchy between the components of a SpinalHDL program. There is multiple way to do this :

- Showing the elements using a hierarchical layout, like in family tree
- Showing the elements using a tree view, like in file explorer
- Showing the elements one by one (multiple diagrams)

3.1.1 Hierarchical layout

The hierarchical layout is the simplest way to achieve the visualization of the hierarchy. The figure 3.2 illustrate a simple hierarchical layout with some children and parent.

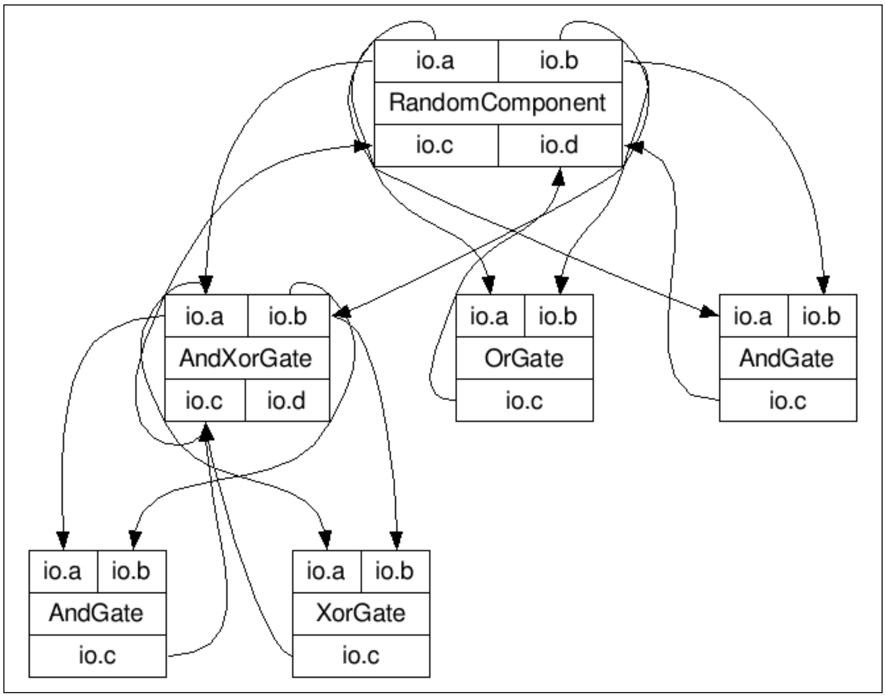

Figure 3.2: The simplest hierarchical visualization, the parent are just representing using the height

The problem by using such a representation is clearly visible, sometimes the ports of a component is an output type port as well as a input type port. This can be seen at the port "io.c" from the `AndXorGate` component.

3.1.2 Tree view

The idea of the tree view is to represent the hierarchical relation like in file explorer. The figure 3.3 show the idea for a SpinalHDL component.

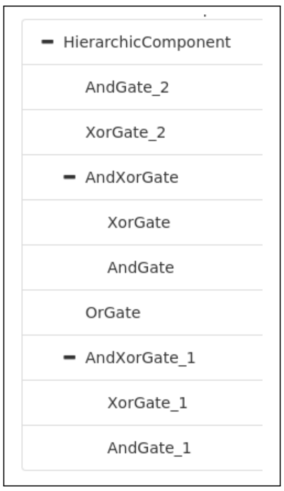

Figure 3.3: A tree view visualization of a SpinalHDL Component, each son are under his parent and eventually possesses it self some sub-Component

The tree view visualization is very usefull to directly detect the relation between some components. The big problem is that we can't see the connections relationship.

3.1.3 Multiple diagrams

As seen with the hierarchical layout figure 3.2, the problem is that sometimes a port is an output for somes components and, in the other way, an input for somes others ones. If we look closely to the problems we could notice that a port receive connections from his brothers and send connections to his childrens.

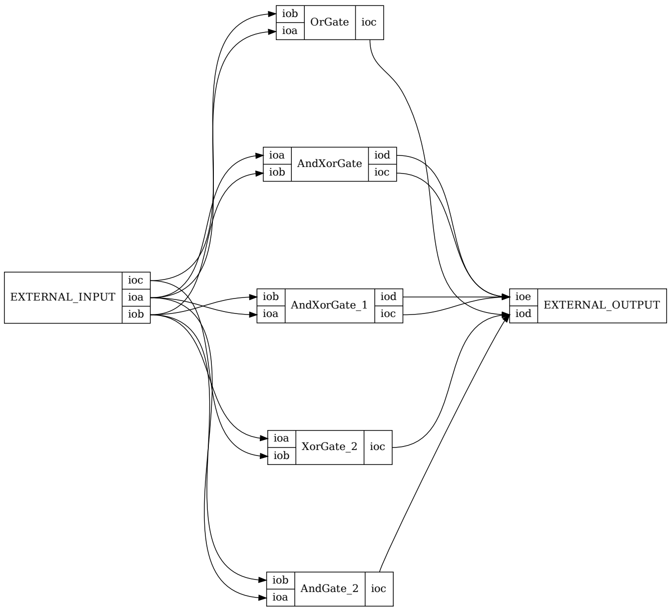

Figure 3.4: Inside of a SpinalHDL components, the components possesses input and output has his own interface and communicates with his children

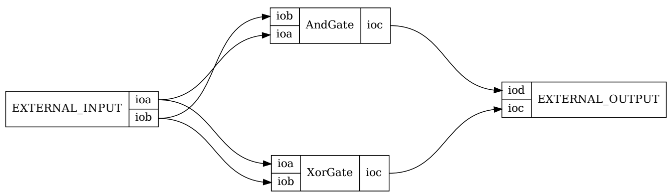

Figure 3.5: Component which is the sub-component of the component from figure 3.4, the external inputs and outputs corresponding to the ones in the parent component

Using such a representation solve the problem discuss in 3.1.1 : if we represent all the components juste using one diagram, sometimes the ports are inputs and sometimes they are outputs. With multiple diagram this problem no more occurs.

The major disadvantages is that we need an interractive diagrams for exploring the children of a components or multiple statics ones which is not practical.

3.2 A visual example

The diagram we want to produce, as explain in 4.3, ownes the following property :

- Oriented
- Cyclic
- Connection between port on nodes and not between nodes

3.3 Model representation

The model representation is builded using the oriented-object paradigm, the figure 3.6 show the class diagram of the models.

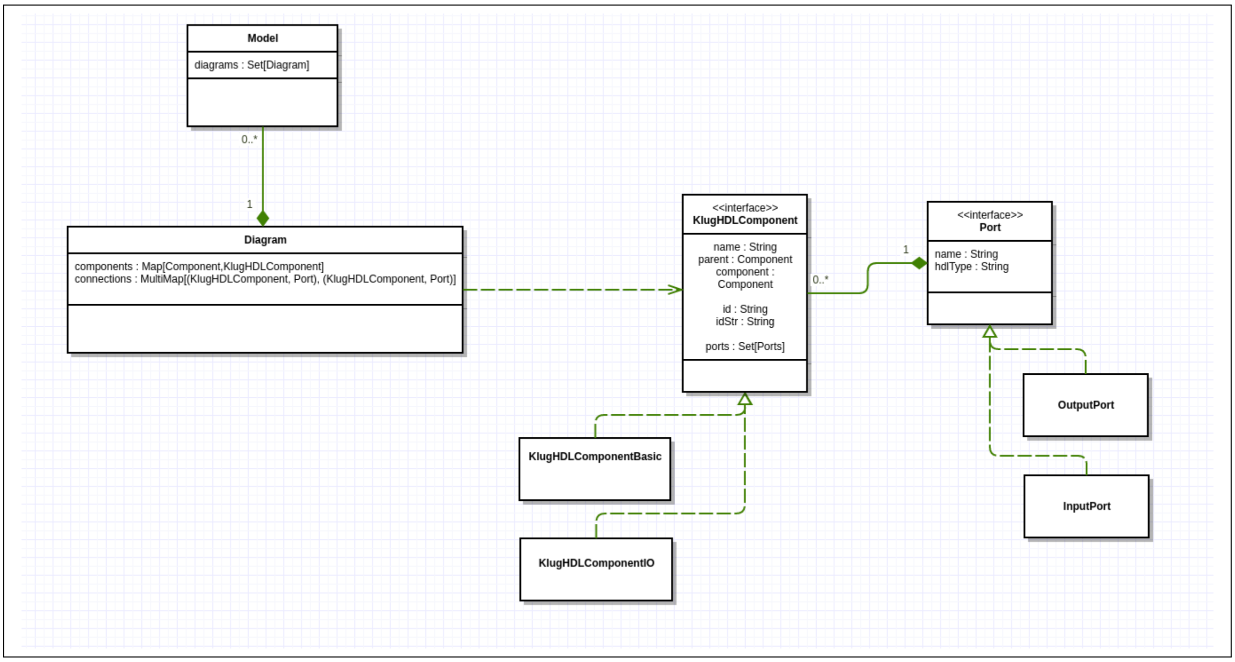

Figure 3.6: The complete class diagram of the intermediate model representation

3.4 Conclusion

The major advantages of such an intermediate representation is the capability to evolve the program in order to produce additionnal output. For now there is just a **dot** and **JSON** backend. It's also provide a way to check the correctness of the diagram after the parsing on the AST : connections with non-existing port for example.

4 Viewing library

This chapter will present some visualization library which could be use to generate and interract with the component diagram. The comparison between those library is based on the features we want to offer to the KlugHDL user (see chapter 1.2). In order to compare those viewing library, we end by discuting the point of the evaluation and the evaluation of all the library mentinnoed.

For the presentation of each of those library for graph visualization, we would use the same graph showed in figure 4.1.

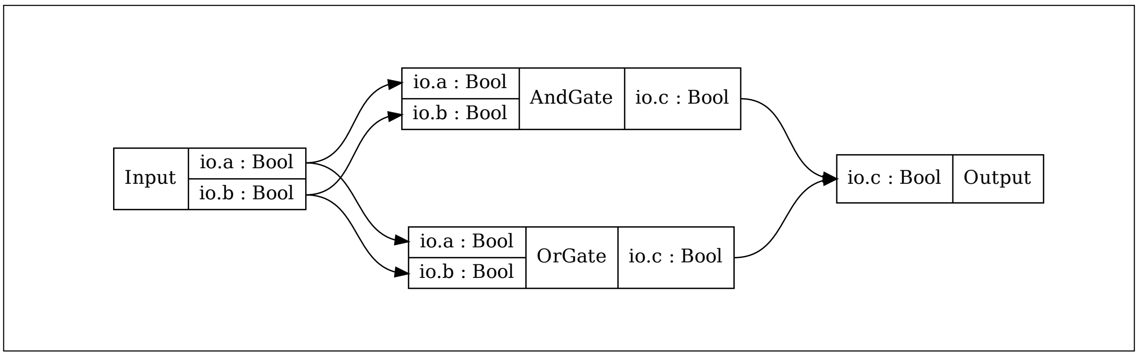

Figure 4.1: The graph we will use in order to compare several Viewing library. This model includes two logical component : a AND and a OR gate and two nodes which are representing the input and output of the parent component

4.1 GraphStream

GraphStream is a Java library for the modeling and analysis of dynamic graphs[3]. The goal of the library is to provide a way to represents graphs and work on it[3]. GraphStream is an active project hosted by the university of Le Havre in France.

4.1.1 Implementation of the base graph model

The figure 4.2 show the base graph model realized with the GraphStream library.

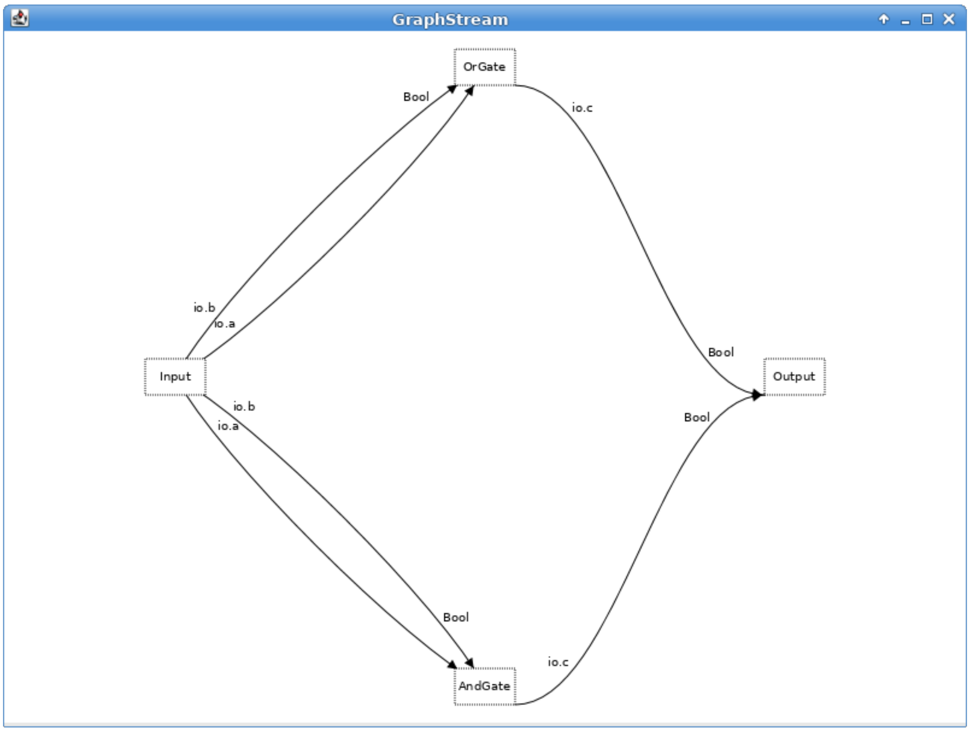

Figure 4.2: The base graph model rendering using graphstream

GraphStream allow us to simply create a graph (or a multigraph) in Java like in listing 3.

```java
Graph graph = new SingleGraph("Example");
graph.addNode("A" );
graph.addNode("B" );
graph.addNode("C" );
graph.addEdge("AB", "A", "B");
graph.addEdge("BC", "B", "C");
graph.addEdge("CA", "C", "A");
```

Listing 3: Modelisation of a simple graph using the GraphStream library

4.1.2 Remarks

With GraphStream, there is no way to control some visual features which are very interresing for hardware component visualisation : the splines of the edges and the anchor position on the node. There is no way to have a node with sub-node like in the base graph model in figure 4.1.

An another special case is the label on the edges. With GraphStream, we could add some label on the edges like in listings 4, but we can't add multiple ones. To add multiple label we need to use sprite like in listing 5.

```java
Graph graph = new MultiGraph("Edges label example");
graph.addAttribute("ui.quality");
graph.addAttribute("ui.antialias");

graph.addNode("A");
graph.addNode("B");
Edge edge = graph.addEdge("AB", "A", "B", true);
edge.addAttribute("ui.label", "A --> B");
graph.display(false);
```

Listing 4: TODO : maybe remove ?

```java
  SpriteManager sman = new SpriteManager(graph);

  // add label 1
  Sprite label1 = sman.addSprite("label1");
  label1.attachToEdge("AB");
  label1.setPosition(0.2, 0, 0);
  label1.addAttribute("ui.label", "Label 1");
  label1.addAttribute("ui.style", "fill-mode:none;");

  // add label 2
  Sprite label2 = sman.addSprite("label2");
  label2.attachToEdge("AB");
  label2.setPosition(0.8, 0, 0);
  label2.addAttribute("ui.label", "Label 2");
  label2.addAttribute("ui.style", "fill-mode:none;");
```

Listing 5: TODO : maybe remove ?

4.1.3 Conclusion

GraphStream ownes some another features which could be usefull to design an application :

- View integration using Swing
- Human-Computer interaction with the view
- A complete CSS interpretation to personalize the nodes and the edges

The main problem using GraphStream is we need to indicates the component information using label on the edges. The figure 4.2 show four nodes and six connections, but if we have a more lot of edges between nodes, it would be difficult to see the type of inputs and outputs, like show in figure 4.3.

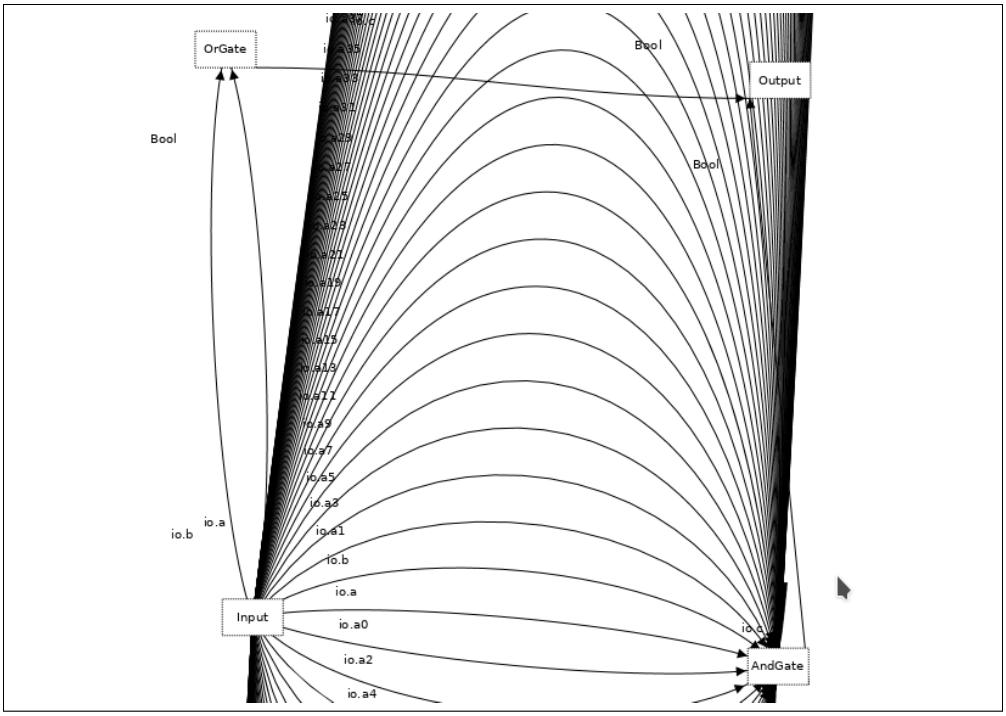

Figure 4.3

In order to overpass these problem, we need to visualize the type in an another form : in the graph 4.1 we have the idea to show the inputs and outputs as a port of the component. With GraphStream we can't do that, so we need to an another library which have these idea of port.

4.2 Draw2D

Draw2D is a HTML5 and Javascript library for visualization and interaction with diagrams and graph[4]. The goal of the library is to provide a way to represent diagrams and graph and manipulate those using the mouse or javascript code. Some existing product already use Draw2d for diagrams manipulation :

- Shape Designer[4]
- BrainBox[4]
- Sankey State[4]

4.2.1 Implementation of the base graph model

Before implementing directly the basic example of the graph 4.1, we need to extend the library with a new component for our usage. In the chapter 4.1.3 we indicates that's the GraphStream library is lacking the idea of port, but Draw2d already have those.

The listing 6 show the realisation of the base graph model example. Note that the object **ComponentShape** and the functions **ComponentShape.addPort()**,**newConnection()** and **createConnection()** are not part of the Draw2d library, it's an extension added for this project.

```
var canvas = new draw2d.Canvas("gfx_holder1");

var andGate = new ComponentShape();
var orGate = new ComponentShape();
var input = new ComponentShape();
var output = new ComponentShape();

canvas.installEditPolicy(new draw2d.policy.connection.DragConnectionCreatePolicy({
  createConnection: createConnection
}));

andGate.setName("AndGate");
orGate.setName("OrGate");
input.setName("Input");
output.setName("Output");

andGate.addPort("io.a", "input");
andGate.addPort("io.b", "input");
andGate.addPort("io.c", "output");

orGate.addPort("io.a", "input");
orGate.addPort("io.b", "input");
orGate.addPort("io.c", "output");

input.addPort("io.a", "output");
input.addPort("io.b", "output");

output.addPort("io.c", "input");

canvas.add(input);
canvas.add(output);
canvas.add(andGate);
canvas.add(orGate);

canvas.add(newConnection(input.getPort("io.a"), andGate.getPort("io.a")));
canvas.add(newConnection(input.getPort("io.b"), andGate.getPort("io.b")));
canvas.add(newConnection(input.getPort("io.a"), orGate.getPort("io.a")));
canvas.add(newConnection(input.getPort("io.b"), orGate.getPort("io.b")));
canvas.add(newConnection(andGate.getPort("io.c"), output.getPort("io.c")));
canvas.add(newConnection(orGate.getPort("io.c"), output.getPort("io.c")));
```

Listing 6: The necessary code to produce the base graph model with the Draw2d library. The object **ComponentShape** and the functions **addPort()**, **newConnection()** and **createConnection()** are not part of the Draw2d library, it's an extension added for this project.

The code 6 is producing the HTML page see in figure 4.4.

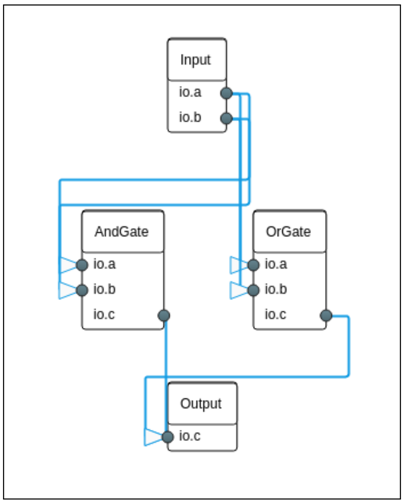

Figure 4.4: Rendering of the base graph model diagram from figure 4.1 using the Draw2D library

4.3 Graph layout

One functionnality the Draw2D don't have is the layout of the diagram. If we said nothing about the position of the node, Draw2D simply overlap all the nodes and manage nothing for the visualization like in figure 4.5.

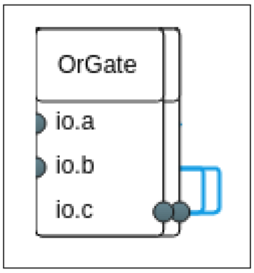

Figure 4.5: When we specify nothing about the node position with Draw2D, the engine just overlaps all the node

In order to produce a visualisable diagram, we need to layout ourself the diagram. We need to mention that's the layout algorithm is a NP-Hard problem in computing theory[5] and for our graph, we have some specialties :

- The graph could be cyclic
- The graph is oriented
- Nodes ownes port, so we need to layout the edges on specific part of the graph

The first two specialties aren't a big deal, but the last one is quiet problematic, he involved a special layout algorithm which take care of the ports position on the nodes. Otherwise the visual result could be chaotics like in figure 4.6, in which we can't see the connection between some specific ports.

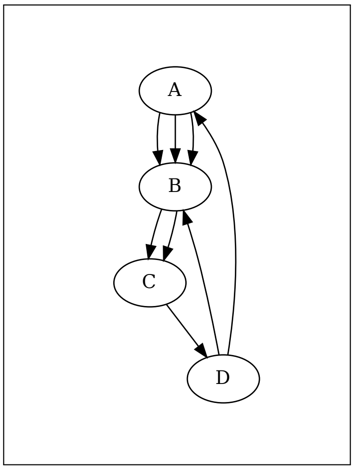

Figure 4.6: Visualization of a multi-graph without port, we could not determine the connection which is representing a certain connection between two component

In order to solve this drawback, we need to layout the graph ourself by using a library. We could also do it ourself by implementing the algorithm for the kind of graph we have, but it's to time-consuming for a project like this.

TODO layout des edges du graph... (questions sur le forum de Draw2d.org)

4.4 Conclusion

GraphStream look like a really good library for manipulating graph and visualize them. The problem is that there is no good way to visualize the ports of the components so we have to labelize the connections and it's look ugly (figure 4.3). Then we have to find a library which allows this idea of ports. We find them with Draw2D and choose it for the project. The disadvantage of using this library is that we can't layout easily the graph by ourself.

5 Parsing the AST

The abstract syntax tree of SpinalHDL is the core components of all the KlugHDL product. He is used to produce the intermediate representation in the generation pipeline (figure 3.1) and once the intermediate representation is built, the rest is easely makable.

The concept of AST as already been explain in chapter 2.2.2. Using this, we are going to see the parsing of the SpinalHDL AST in order to generate an intermediate model. The parsing and the following generation into the intermediate model is separate in multiple phases :

- Diagram parsing
- Components parsing
- Ports parsing
- Connections parsing

As illustrate in the intermediate model class diagram 3.6, the intermediate representation is composed like the following :

- A model possesses multiple diagrams (at least two : the toplevel and the root component)
- A diagrams possesses at least one component with zero or more ports
- A diagrams possesses zero or more connections which are

A model could be represented, using scala, with the class declaration in listing 7. A model is a set of diagrams which are related to a toplevel component.

```
case class Model(topLevel : Component) {

  var diagrams : Set[Diagram] = Set()
}
```

Listing 7: Declaration of the model with scala, a model is basically a set of diagrams and is attached to a topLevel component, which is the only component of the AST which as no parent

5.1 Diagram parsing

Generating a diagram (see listing 8) is trivial because we just have to retrieve all the components which are parents, the algorithm is implemented in listing 9.

```
class Diagram(val parent : Component) {

  var components : Map[Component, KlugHDLComponent] = Map()
  var connections : mutable.MultiMap[
      (KlugHDLComponent,Port),
      (KlugHDLComponent, Port)
    ]
  }
```

Listing 8: Declaration of a the diagram class with scala, a diagram is a set of components (here as a Map) and a set of connections (here as a multimap for the orientation)

```
def generateDiagrams(component : Component) : Unit = {
  diagrams += new Diagram(component.parent)
  component.children.foreach(generateDiagrams)
}
```

Listing 9: This function parse the AST and generate all the corresponding diagrams object for a specific component

Because we are using a set, we don't have to check if the digrams already exist. To make this possible we have to redifine the `equals` function like in listing 10.

```
override def equals(o : scala.Any) : Boolean = o match {
  case d : Diagram => this.parent == d.parent
  case _ => super.equals(o)
}
```

Listing 10: We have to override the equals function in order to use the diagrams in a set as expected

5.2 Components parsing

The parsing and generation of the components is made for each diagrams one by one. Here, the algorithm is quite simple too. The algorithm is implemented in the listing 12 and the KlugHDLComponent is declared in listing 11.

```
sealed trait KlugHDLComponent {

  val name : String
  val parent : Component
  val component : Component = this match {
    case KlugHDLComponentBasic(_, c, _) => c
    case KlugHDLComponentIO(_, _) => null
  }
}

final case class KlugHDLComponentBasic(name : String, override val component : Component,
 ↪  parent : Component) extends KlugHDLComponent

final case class KlugHDLComponentIO(name : String, parent : Component) extends
 ↪  KlugHDLComponent
```

Listing 11: Declaration of the Components class (here names KlugHDLComponents). There is two type of components : the basic ones and the ones who represent the external world input and output like in figure 3.5

```scala
private def generateComponent(diagram : Diagram) : Model = {
  diagram.foreachChildren(diagram.addComponents, topLevel)
  if (diagram.parent != null)
    diagram.addIoComponents(diagram.parent)
  this
}

def addComponents(component : Component) : Unit = {
  components += (component -> KlugHDLComponentBasic(component.definitionName, component,
   ↪   component.parent))
}

def addIoComponents(component : Component) : Unit = {
  if (component == null)
    components += (component -> KlugHDLComponentIO("NULL", component))
  else
    components += (component -> KlugHDLComponentIO(component.definitionName, component))
}
```

Listing 12: Components parsing and generation in scala for one diagram

Note that we generate the interface for the external world connections in the model. This two KlugHDLComponentIO aren't present in the SpinalHDL AST.

5.3 Ports parsing

The ports generation is, again, easy to make. In the SpinalHDL model, the Components class already ownes a `getAllIo` function which return all the inputs and outputs node for a components. So we just need to create a ports object (listing 13) from these and add it to the correct KlugHDLComponents. The algorithm is implement in the listing 14.

```scala
sealed trait Port {

  val name : String
  val hdlType : String
}

final case class InputPort(name : String, hdlType : String) extends Port

final case class OutputPort(name : String, hdlType : String) extends Port
```

Listing 13: Declaration if the Port class with scala, there is two types of ports : input and output

```scala
def generatePort(diagram: Diagram): Unit = {
  def generatePort(entry: (Component, KlugHDLComponent)): Unit = {
    if (entry._1 != null) {
      entry._1.getAllIo.foreach { bt =>
        entry._2.addPort(Port(bt))
      }
    }
  }

  diagram.components.foreach(generatePort)
}
```

Listing 14: Ports parsing and generation in scala for one diagrams, the ports are generating component by component

5.4 Connections parsing

The connections parsing is the hardest implementation to do. The reason is we have multiple type of connections :

- input connections
- output connections
- input connections from parent
- output connections from parent

In addition, to find the staring point of a connection, we have to parse recursively all the nodes of the AST until we found a `Basetype` object which is the type for the inputs and outputs nodes of a SpinalHDL component.

5.4.1 Input connections

The input connections represent all the connections generated romom the inputs of a nodes. Firstable, we have to retrieve all the inputs for a specific AST nodes then we need to generate a connections between two ports on two nodes. The implementation could be seen in listing 15.

```
def parseInputConnection(component: Component): Unit = {

  def parseInputs(node: Node): List[BaseType] = {

    // iterate over all the inputs on each nodes of the AST until founding an
    // output basetype
    def inner(node: Node): List[BaseType] = node match {
      case bt: BaseType =>
        if (bt.isOutput) List(bt)
        else List(bt) ::: node.getInputs.map(inner).foldLeft(List(): List[BaseType])(_ :::
          ↪  _)
      case null => List()
      case _ => node.getInputs.map(inner).foldLeft(List(): List[BaseType])(_ ::: _)
    }

    inner(node).filter {
      _ match {
        case bt: BaseType => bt.isOutput
        case _ => false
      }
    }
  }

  for {
    io <- component.getAllIo
    if io.isInput
    input <- parseInputs(io)
  } {
    diagram.addConnection(input.component, Port(input), io.component, Port(io))
  }
}
```

Listing 15: Implementation in scala of the parsing and generation for all the inputs connections for a specific component

5.4.2 Output connections

The output connections represent all the connections generated from the outputs of a node. The problem is the same as for the input connections but in the other way : we need to parse the consumer for a specific node like in listing.

```
def parseOutputConnection(component: Component): Unit = {

  def parseConsumers(node: Node): List[BaseType] = {

    // iterate over all the inputs on each nodes of the AST until founding an
    // input basetype
    def inner(node: Node): List[BaseType] = node match {
      case bt: BaseType =>
        if (bt.isInput) List(bt)
        else List(bt) ::: node.consumers.map(inner).foldLeft(List(): List[BaseType])(_ :::
          ↪ _)
      case null => List()
      case _ => node.consumers.map(inner).foldLeft(List(): List[BaseType])(_ ::: _)
    }

    inner(node).filter {
      _ match {
        case bt: BaseType => bt.isInput
        case _ => false
      }
    }
  }

  for {
    io <- component.getAllIo
    if io.isInput
    consumer <- parseConsumers(io)
  } {
    diagram.addConnection(io.component, Port(io), consumer.component, Port(consumer))
  }
}
```

Listing 16: Implementation in scalaof the parsing and generation for all the inputs connections for a specific component

5.4.3 Parent connections

Finally we could parse and generate the connections with the parent. This part is slightly easier than the two previous one. Because we just have to generate the inputs and consumers and subtract them like in a set. The algorithm is describe in listing 17.

```
def parseParentConnection(component: Component): Unit = {

  def parseInputParentConnection(component: Component): Unit = {
    if (component.parent != null) {
      val con = for {
        io_p <- component.parent.getAllIo
        io <- component.getAllIo
        if io.getInputs.contains(io_p)
      } yield (io_p.component, Port(io_p), io.component, Port(io))
      con.foreach(diagram.addConnection)
    }
  }

  def parseOutputParentConnection(component: Component): Unit = {
    if (component.parent != null) {
      val con = for {
        io_p <- component.parent.getAllIo
        io <- getInputs(io_p)
        if io != null
      } yield (io.component, Port(io), io_p.component, Port(io_p))
      con.foreach(diagram.addConnection)
    }
  }

  def getInputs(bt: BaseType): List[BaseType] = {
    val comp = bt.component

    def inner(n: Node, acc: List[Node]): List[Node] = {
      if (n == null) acc
      else if (n.component != comp) {
        acc
      }
      else if (n.getInputsCount > 1) {
        n.getInputs.flatMap(n => inner(n, Nil)).toList
      }
      else {
        inner(n.getInputs.next(), n :: acc)
      }
    }

    inner(bt, Nil).map(_.getInputs.next().asInstanceOf[BaseType])
}

  parseInputParentConnection(component)
  parseOutputParentConnection(component)
}
```

Listing 17: Implementation in scala of the parsing and generation of all the connections (inpus and outputs ones) with the parent

5.5 Conclusion

The parsing of the AST is the hardest part to understand, there is a lot to know about SpinalHDL itself and a large amount of special case to understand. By the way, it's the most important part too, with the generation of the intermediate model we could do all the rest.

6 Diagram visualization

This chapter will present the visualization of the diagram. First, we will introduce the intermediate model backend use by KlugHDL in order to produce an output for further use. Then we will see the static and dynamic visulalization which are available with KlugHDL.

6.1 Intermediate representation

The intermediate representation generate after the AST parsing is an object-oriented representation created for KlugHDL. In order to use the generated diagrams in other software we need to produce a standard output. In KlugHDL those additional output are called "backend".

6.1.1 The DOT backend

DOT and Graphviz are two software for diagram visualization. The diagam is written using the "dot" language and then is compiled to various image format like png or pdf.

The advantage of using DOT is the simplicity. The figure 6.1 show a graphviz diagram generated by the DOT program. The corresponding code is available in listing 6.2.

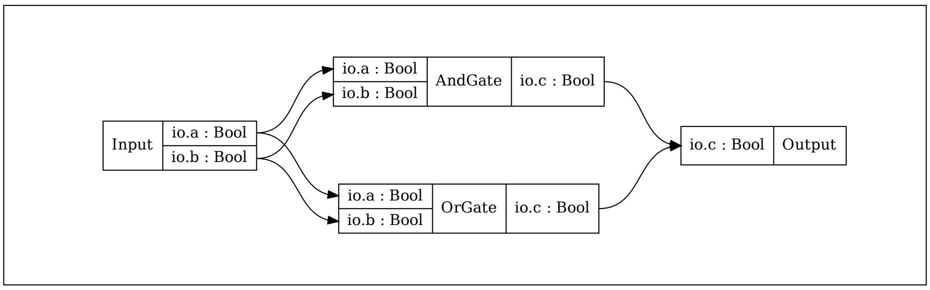

Figure 6.1: A complete graphviz diagram generated by DOT, the code of this example is available in listing 6.2

```
digraph g {
  node [shape=record];
  graph [rankdir=LR,ranksep="1",nodesep="1"];
  AndGate [label="{{<a>io.a : Bool|<b>io.b : Bool}|AndGate|{<c>io.c : Bool}}"];
  OrGate [label="{{<a>io.a : Bool|<b>io.b : Bool}|OrGate|{<c>io.c : Bool}}"];
  Input [label="{Input|{<a>io.a : Bool|<b>io.b : Bool}}"];
  Output [label="{{<c>io.c : Bool}|Output}"];
  Input:a -> AndGate:a;      Input:b -> AndGate:b;
  Input:a -> OrGate:a;       Input:b -> OrGate:b;
  OrGate:c -> Output:c;      AndGate:c -> Output:c;
}
```

Figure 6.2: A complete example of a DOT program, the corresponding diagram is available in figure 6.1

We could just notice that we use the `shape=record` option of DOT in order to genereate the ports on the nodes and link them with the connection.

6.1.2 The JSON backend

JSON (JavaScript Object Notation) is a format mostly use by javascript for object serialization. The format is completely include in the javascript language which is able to directly understand the object notation wihtout any additional code to write.

At the beginning of the project, we try to produce the javascript output directly from the intermediate model. The problem was [illegible]e change the library we would like to use (Draw2d in this case), we have to change the entire backend. The solution is to produce a JSON output, whatever which library we are using, the model would always be the same.

For this backend we use the lift json library [6]. The library is very easy to use and offer a small DSL in order to write readable code. The listing 18 show an example of the lift json library to create a KlugHDL component in JSON.

```
def generateJson(klugHDLComponent: KlugHDLComponent): JValue = klugHDLComponent match {
  case KlugHDLComponentBasic(name, _, _) =>
    ("name" -> name) ~
    ("type" -> "default") ~
    ("ports" -> klugHDLComponent.ports.map(generateJson))
  case KlugHDLComponentIO(name, _) =>
    ("name" -> name) ~
    ("type" -> "io") ~
    ("ports" -> klugHDLComponent.ports.map(generateJson))
}
```

Listing 18: The lift json library offer the opportunity to write readable and scalable code with her DSL

6.2 Static visualization

With KlugHDL it's very simple to generate diagram in a pdf format. For example, this could be usefull for a report documentation. The listing 19 show how to do it with a complete example.

```
/**
 * Example for generating pdf diagam with dot
 * of a SpinalHDL component
**/

 object DotExample extends App {

  // Create the vhdl rtl
  val report = SpinalConfig(targetDirectory = "vhdl").generateVhdl(new SmallComponent)

  // generate the diagram
  Dot(targetDirectory = "dot").generatePDFDiagram(Model(report.toplevel))
}
```

Listing 19: A complete KlugHDL example on how to generate a pdf diagram using the Dot backend

6.3 Dynamic visualization

A dynamic diagram visualization is also provide with KlugHDL. The listing 20 show how to generate the json model for a SpinalHDL component.

```
/**
 * Example for generating the json model
 * of a SpinalHDL component
 **/

 object JsonExample extends App {

   // Create the vhdl rtl
  val report = SpinalConfig(targetDirectory = "vhdl").generateVhdl(new SmallComponent)

   // generate the model
   Json(targetDirectory = "json").generateJsonModel(Model(report.toplevel))
 }
```

Listing 20: A complete KlugHDL example on how to generate a json model

6.3.1 Extending the draw2d library

We have to extends the draw2d library in order to create the specific diagram we want. The only addition to make is to create an another node for draw2d. This is done by extending the `VerticalLayout` shape. The whole new shape is created with the code in listing 21.

The whole code for parsing the json model and display it into a web page is available in the source code of the project.

6.3.2 Problems

For the moment, there is a problem with the navigation through the diagrams. The navigation from parent to brothers is fine but in the other direction, from the children to the parent, some component are, sometimes, calling the callback method on the double click input multiple times. This create, first, a very long callback and then corrupt the memory and raise exception.

```
ComponentShape = draw2d.shape.layout.VerticalLayout.extend({
  NAME: "ComponentShape",
  init: function (attr) {
      var _this = this;
      this._super($.extend({
          bgColor: "#ffffff",
          color: "#000000",
          stroke: 1,
          radius: 3
      }, attr));
      this.classLabel = new draw2d.shape.basic.Label({
          text: "ClassName",
          stroke: 1,
          fontColor: "#000000",
          bgColor: "#ffffff",
          radius: this.getRadius(),
          padding: 10,
          resizeable: true,
          editor: new draw2d.ui.LabelInplaceEditor(),
          onDoubleClick: function () {
              _this.doubleClickCallBack()
          }
      });
      this.add(this.classLabel);
  },
  setName: function (name) {
      this.classLabel.setText(name);
  },
  getName: function () {
      return this.classLabel.text;
  },
  getPort: function (name) {
      return this.getPorts().find(function (entry) {
          return entry.name == name
      });
  },
  addPort: function (name, type) {
      var _this = this;
      var label = new draw2d.shape.basic.Label({
          text: name,
          stroke: 0,
          radius: 90,
          bgColor: null,
          padding: {left: 10, top: 3, right: 10, bottom: 5},
          fontColor: "#000000",
          resizeable: true,
          onDoubleClick: function () {
              _this.doubleClickCallBack()
          },
          editor: new draw2d.ui.LabelEditor()
      });

      var port = label.createPort(type);
      port.setName(name);

      this.add(label);

      return label;
  },
  doubleClickCallBack: function () {
      console.log("double click callback")
  },
});
```

Listing 21: A new prototype in draw2d is created by extending the VerticalLayout shape, then we have to provide a function for creating additional stack element in the shape

7 Conclusion

This project have try to offer an additionnal tool for SpinalHDL in order to increase his the popularity. Domain specific language are more and more use in computer software this day and KlugHDL could be describe as a tool for devellopment and documentation.

Writing VHDL code over and over again seems to be painful and SpinalHDL try to make the developper life easier, but VHDL ownes a lot of tools to increase productivity. KlugHDL is the first "tools" of the whole world of SpinalHDL and launch a new era for SpinalHDL and his believer.

KlugHDL is able to parse and generate static diagram with dot and Graphviz for any simple SpinalHDL components. Components with bus or complex basetype, like $Uint(nbits)$ aren't supported for the moments. KlugHDL is also here to demonstrate it's possible to generate diagram from the SpinalHDL AST.

We have seen that parsing the AST could be tricky and sometimes really strange. We need additionnal SpinalHDL Components and a lot deeper understanding of the SpinalHDL AST in order to parse and generate all the possible components which could be written.

TODO : end this...

Bibliography

[1] Wikipedia, VHDL, Online; accessed 05.10.2016, **2016**.

[2] C. Papon, SpinalHDL : About SpinalHDL, Online; accessed 05.10.2016, **2016**.

[3] T. G. Team, GraphStream : a Dynamic Graph Library, Online; accessed on 09.10.2016, **2015**.

[4] A. Herz, Draw2D touch, **2007**, `http://draw2d.org/` (visited on 03/25/2014).

[5] R. Tamassia, *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*, Chapman & Hall/CRC, **2007**.

[6] L. W. Team, Parsing and formatting utilities for JSON, Online; accessed on 11.12.2016, **2016**.

Appendices

A Facts sheet : Generate a component diagram

Generate a component diagram

1 : Identification summary

Abstract : The user want to generate the component diagram from the code
Authors : Sylvain Julmy

Actors : SpinalHDL user
Creation date : 12.10.2016 **Version** : 0.1
Modification date : 18.10.2016 **Responsible** : Sylvain Julmy

2 : Sequence description

Precondition : The user activate the compilation options

Standard progress :

1. The user indicate to the compiler to launch KlugHDL
2. The user compile the program using the standard Scala compiler
3. A Windows appear and show the component diagram

Alternative sequence :
A1 : Start at point 2 of standard progress

1. There is a compilation error
2. Nothing append by KlugHDL
3. Restart at point 1 of standard progress

Postcondition : The component diagram is displayed

3 : HCI needing (Optional)

A windows that's show the component diagram

4 : Complementary remarks

KlugHDL is not really a standalone application, it's more like an option of SpinalHDL at the compilation.

B Facts sheet : Display, hide and toggle the view of the edge's type

Display, hide and toggle the view of the edges's type

1 : Identification summary

Abstract : The user want to see the types of some edges
Authors : Sylvain Julmy

Actors : SpinalHDL user
Creation date : 12.10.2016
Modification date : 14.10.2016
Version : 0.1
Responsible : Sylvain Julmy

2 : Sequence description

Precondition : The diagram is generated and open.

Standard progress :

1. The user click on "Show types"
2. The diagram display the types of the edges

Postcondition : The types of the selected edges are display.

3 : HCI needing (Optional)

The system needs the following HCI element :

- A toggable button "Show types"

4 : Complementary remarks

All the edges are affected by this operation

C Facts sheet : Enter a component

Enter a component

1 : Identification summary

Abstract : The user want to enter inside a component (zoom in) in order to show the sub-component

Authors : Sylvain Julmy

Actors : SpinalHDL user

Creation date : 12.10.2016 **Version** : 0.1

Modification date : 18.10.2016 **Responsible** : Sylvain Julmy

2 : Sequence description

Precondition : The diagram is generated and open.

Standard progress :

1. The User double click on a component
2. The system display the inside of the component

Postcondition : The double clicked component is maximize and is now the "root" component of the window.

3 : HCI needing (Optional)

- A visualization of the component and the sub-component

4 : Complementary remarks

Except the leaf component which has no sub-component and the BlackBox, every component are zoomable.

D Facts sheet : Exit a component

Exit a component

1 : Identification summary

Abstract : The user want to exit (zoom out) a component
Authors : Sylvain Julmy

Actors : SpinalHDL user
Creation date : 12.10.2016
Modification date : 18.10.2016
Version : 0.1
Responsible : Sylvain Julmy

2 : Sequence description

Precondition : The user has entering a component previously.

Standard progress :

1. The user zoom out (using the mouse wheel)
2. The parent component is displaying by the system

Exception sequence :
E1 : Start at point 1 of standard progress

1. The is no parent component for the current one
2. Nothing is changing
3. Case is ended

Postcondition : The user view the inside of the parent component.

3 : HCI needing (Optional)

- A visualization of the component and his parent

4 : Complementary remarks

Except the root component, each component could be zoom out.

E Facts sheet : View a component diagram

View the component diagram

1 : Identification summary

Abstract : The user want to display and view the component diagram of his program
Authors : Sylvain Julmy

Actors : SpinalHDL user
Creation date : 12.10.2016 **Version** : 0.1
Modification date : 21.10.2016 **Responsible** : Sylvain Julmy

2 : Sequence description

Precondition : The RTL has been generated without compilation error and the user has enter the correct option for the compilation.

Standard progress :

1. The user compile is program
2. The diagram is displayed by the system

Postcondition : The diagram is display.

3 : HCI needing (Optional)

- A diagram of the component of the user program

4 : Complementary remarks

The diagram could be generated but not necessarily display by the user.

F Facts sheet : Display, hide, toggle the hierarchical tree view

Display, Hide and Toggle the hierarchical tree view

1 : Identification summary

Abstract : The user want to hide or display the hierarchical tree view
Authors : Sylvain Julmy

Actors : SpinalHDL user
Creation date : 12.10.2016 **Version** : 0.1
Modification date : 21.10.2016 **Responsible** : Sylvain Julmy

2 : Sequence description

Precondition : The diagram is generated and open.

Standard progress :

1. The user click on the "Hierarchical tree view" button
2. The hierarchical tree view is display or hide

Postcondition : The hierarchical view is visible if she was previously hidden and is not visible if she was previously display.

3 : HCI needing

- A tree view of the componenent and their sub-component
- A toggable button "Hierarchical tree view"

4 : Complementary remarks

G Facts sheet : Generate the RTL

Generate the RTL

1 : Identification summary

Abstract : The user want to generate the RTL of his SpinalHDL program
Authors : Sylvain Julmy

Actors : SpinalHDL user
Creation date : 12.10.2016 **Version** : 0.1
Modification date : 21.10.2016 **Responsible** : Sylvain Julmy

2 : Sequence description

Precondition :

Standard progress :

1. The user compile his code using the command `sbt compile`
2. The RTL is generated

Exception sequence :
E1 : Start at point 1 of standard progress

1. There is a compilation error
2. Case is ended

Postcondition : The RTL is generated

3 : HCI needing (Optional)

There is no HCI needing because everything is done by SBT.

4 : Complementary remarks

Generate the RTL is not a job done by KlugHDL but by SpinalHDL.

H Facts sheet : Filter the view

Filter the view

1 : Identification summary

Abstract : The user want to filter the view (including the hierarchical tree view) to see a specific kind of component

Authors : Sylvain Julmy

Actors : SpinalHDL user

Creation date : 12.10.2016 **Version** : 0.1

Modification date : 21.10.2016 **Responsible** : Sylvain Julmy

2 : Sequence description

Precondition : The diagram is generated and open.

Standard progress :

1. The user enter the text specific to a component (class name)
2. The view turn gray the component who arent matching the text

Postcondition : The element who aren't matching the introduce pattern are grayscale.

3 : HCI needing (Optional)

- A writable textfield

4 : Complementary remarks