MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Information and Communication Technologies (ICT)

KlugHDL : a SpinalHDL diagram generator

Author:
Julmy Sylvain

Under the direction of:
Dr. Mudry Pierre-André
Institut Systèmes Industriels

Lausanne, HES-SO//Master, January 26, 2017

Contents

1 Introduction

A program is a sequence of machine instructions which are written by a human and executed by a computer. The VHDL language 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 offer a way to outmatch those disadvantages. The main goal of a DSL (domain-specific language) is to offer a more pleasant way to code and that's what does SpinalHDL. As written before, the code is written by a human and by using SpinalHDL, the human becomes happier than using just the VHDL.

KlugHDL is another tool to make the human happier, it's a component diagram generator which produces statics and dynamics diagrams.

1.1 Context

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

- No restriction to the genericity of 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 advantages.
- 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 listing 1.1 shows a AND gate written with SpinalHDL with the corresponding generated VHDL code. We can see the similarity between the two codes.

```
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
}
```

```
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 1.1: 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 will analyse a SpinalHDL program in order to produce a block diagram of the corresponding hardware description. This application should offer the following activities :

- Displaying the complete graph of the program
- Navigating through all the hierarchical levels of the program (Components inside another component)
- Displaying signal's information on the edges :
  - type
  - input or output
  - ...
- Diplaying a hierarchical view (like in filesystems or in Quartus)
- Offering a filter functionality in order to view only some specifics components

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

2 Analysis

In this chapter, we would discuss about the "What do we want ?" questions. It means what does 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 the whole implementation or conception, but a small part is useful for this project. Finally, we introduce the concept of the SpinalHDL's 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 wants to do with KlugHDL. The figure 2.1 shows the use case of KlugHDL.

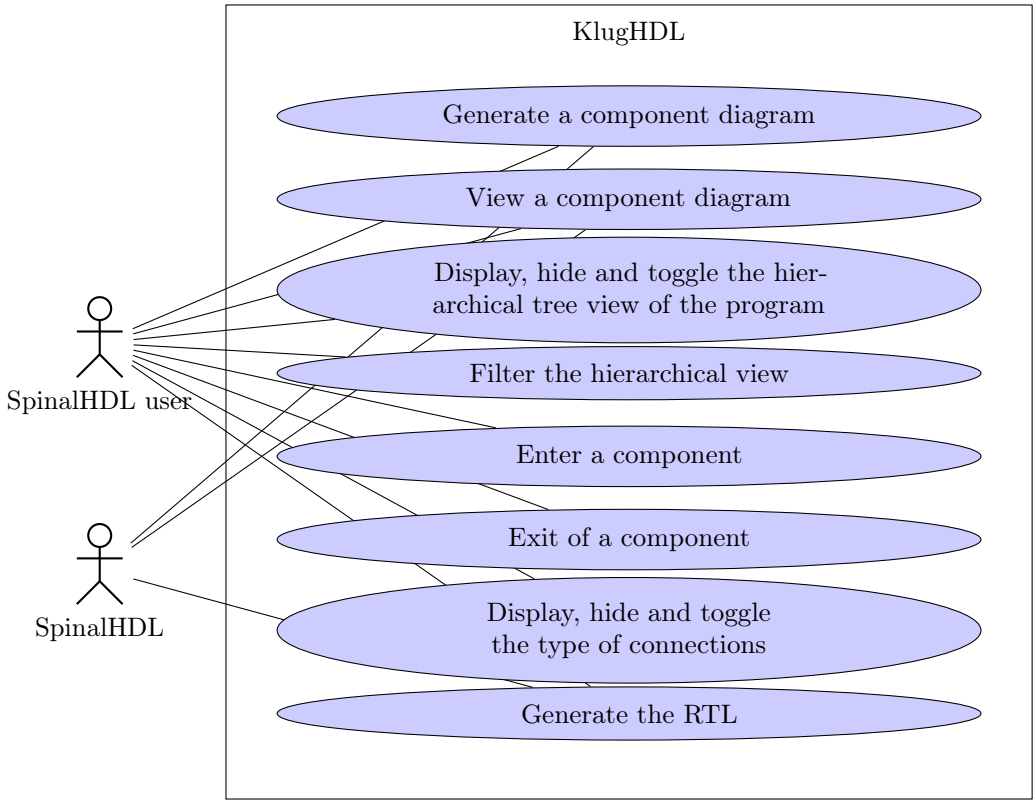

Figure 2.1: Use case of KlugHDL

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

- 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 connection's type.

2.2 SpinalHDL

As explained in chapter 1, SpinalHDL is a DSL for VHDL programing. SpinalHDL runs with various internals constructions such as an AST or a Component Tree. Those are the major elements 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 1.1. Like in Verilog and VHDL, we can define components that could be used to build a design hierarchy. But unlike them, we don't need to bind them at instantiation [2]. That's why there is just one component in SpinalHDL versus the architecture and entity combination in VHDL.

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

- a parent-children relationship
- a brother relationship

A component could contain and communicate with other components, that's the parent-children relationship, and a component that could communicate with the component contained by its parent, is a brother's relationship. Those two relations could be seen in listing 2.1. The component communicates with its children which are communicating together.

A component may have other elements inside it 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 nodes are marked by operator and the leaves are marked by the operand of those operators. For example, the expression $5 * 2 + 3$ has its equivalent AST shown in figure 2.2.

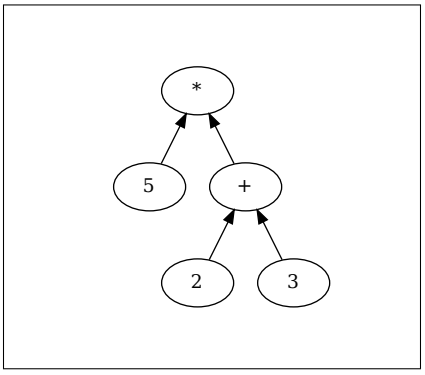

Figure 2.2: AST of the expression $5 * 2 + 3$.

The understanding of the AST is the main concept of this project. With it, we are able to generate the whole model for further implementations.

```
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 2.1: There are two different types of connections with SpinalHDL, from brother to brother or from a parent to one of its children. The corresponding diagram to this code is shown in figure 3.2

3 Diagrams modeling

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

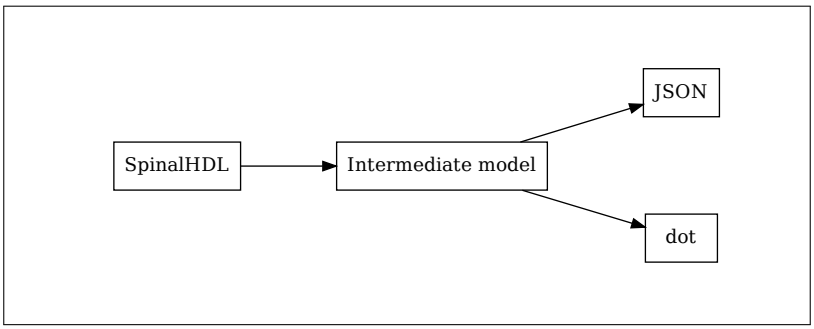

Figure 3.1: Representation of the entire generation pipeline used by KlugHDL in order to generate different outputs. The intermediate model is used to produce the target code, for example DOT or JSON file.

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

3.1 Hierarchy visualization

The diagrams produced by KlugHDL are of a special kind, they are hierarchical. That means we need to find a way to show the hierarchy between the components of a SpinalHDL program. There are multiple ways 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 illustrates a simple hierarchical layout with some children and parents.

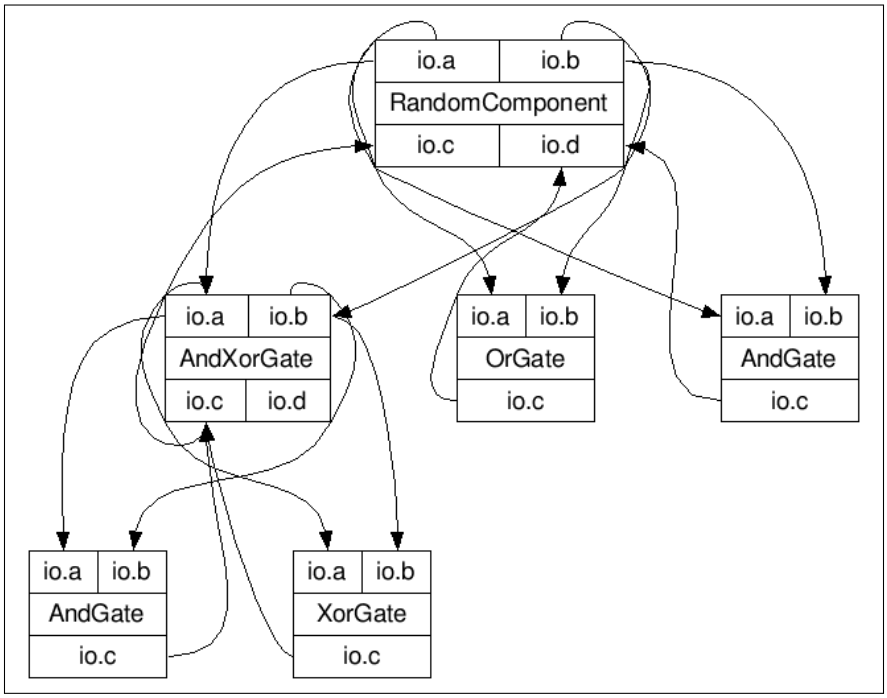

Figure 3.2: The simplest hierarchical visualization, where parents are just represented using various heights

The problem by using such a representation is clearly visible, sometimes the ports of a component is an output type port as well as an input type port. This can be seen at the port "io.c" from the `AndXorGate` component. In listing 3.1, we can see that the value `io.c` is written as an output of the component. In the figure 3.2, the same port is the source and the target of some connections. This can cause some misunderstanding.

```
class AndXorGate extends Component {

  val xorGate = new XorGate
  val andGate = new AndGate

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

  xorGate.io.a := io.a
  xorGate.io.b := io.b
  andGate.io.a := io.a
  andGate.io.b := io.b

  io.c := xorGate.io.c
  io.d := andGate.io.c
}
```

Listing 3.1: The AndXorGate component written with SpinalHDL. We can see that the "c" port is an output of the components and in figure 3.2 the port is shown as an input port and as an output port.

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 depicts the idea for a SpinalHDL component.

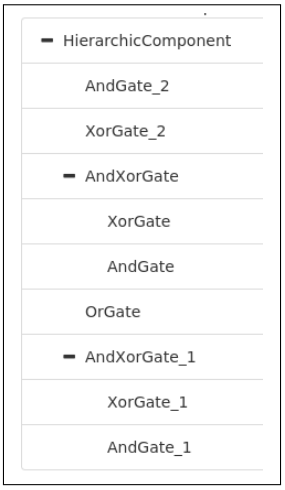

Figure 3.3: A tree view visualization of a SpinalHDL component, each child is under its parent and eventually possesses itself some sub-components.

The tree view visualization is very useful to directly detect the relation between some components. The big problem is that we can't see the connection relationships.

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 some components and, in the other way, an input for some other ones. If we look closely to the problem we could notice that a port receives connections from its brothers and sends connections to its childrens.

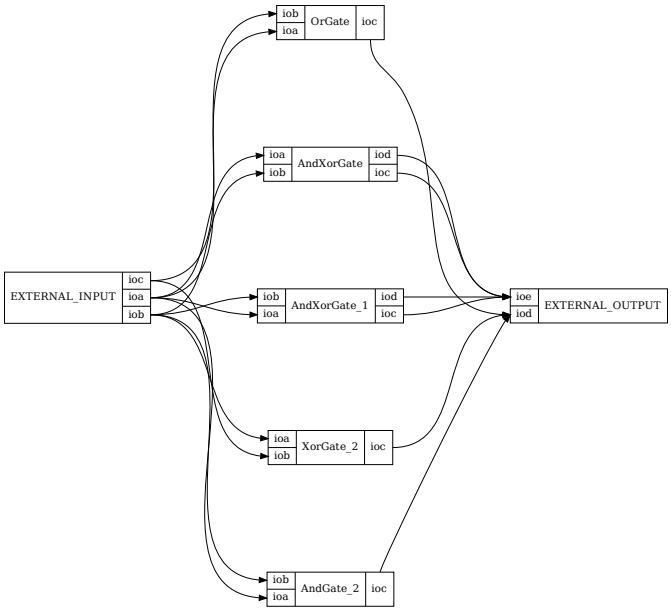

Figure 3.4: Inside a SpinalHDL component, the component owns inputs and outputs on his own interface and communicates with its children

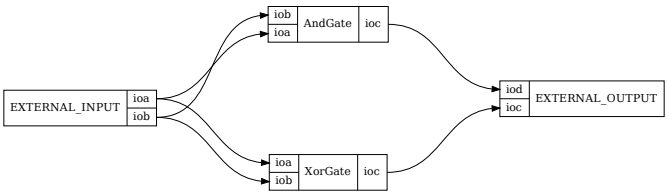

Figure 3.5: Component which is the sub-component of the component from figure 3.4, the external inputs and outputs are corresponding to the ones in the parent component

Using such a representation solves the problem discussed in 3.1.1 : if we represent all the components just using one diagram, sometimes the ports are inputs and sometimes they are outputs. With multiple diagrams this problem does not occur anymore.

The major disadvantage is that we need an interactive diagram for exploring the children of a component or multiple static ones which is not practical.

3.2 A visual example

The diagram we want to produce, as explained in chapter 4.3, owns the following properties :

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

3.3 Model representation

The model representation is built using the object-oriented paradigm. The figure 3.6 shows the class diagram of the models.

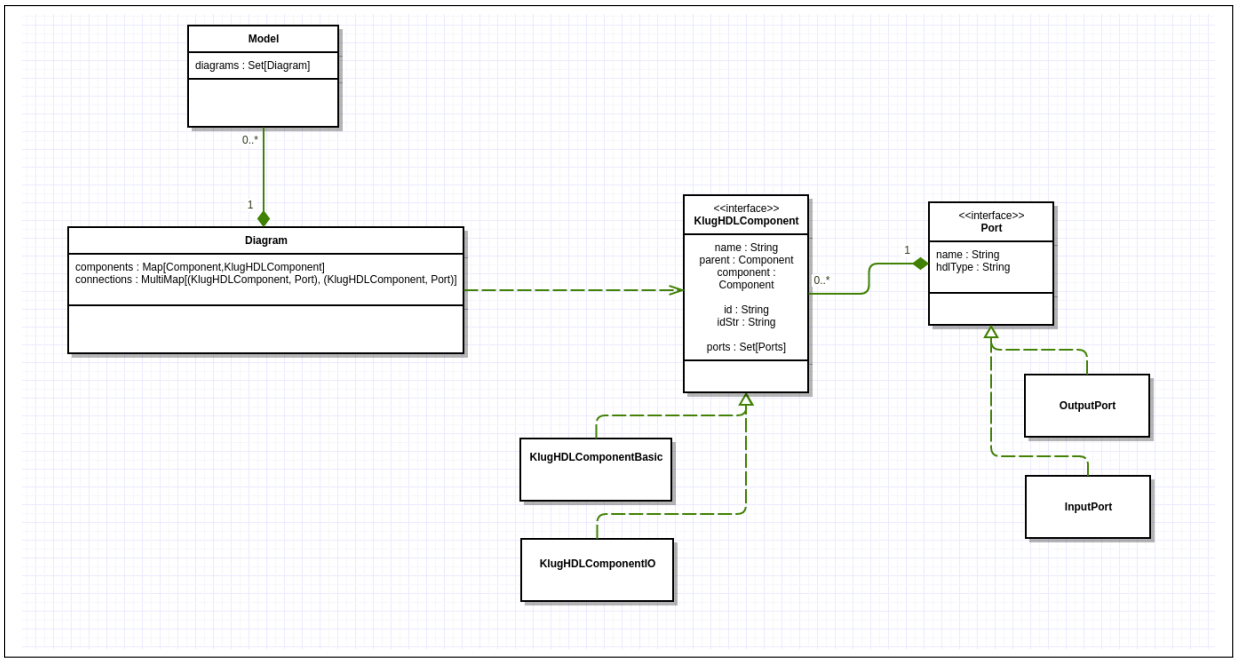

Figure 3.6: The complete class diagram of the intermediate model representation

3.4 Conclusion

The major advantage of such an intermediate representation is the capability to evolve the program in order to produce additional outputs. For now there is just a **DOT** and **JSON** backend. It also provides 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 libraries which could be used to generate and interact with the components diagram. The comparison between those libraries is based on the features we want to offer to the KlugHDL user (see chapter 1.2). In order to compare those viewing libraries, we end by discussing the evaluation of all the libraries mentioned.

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

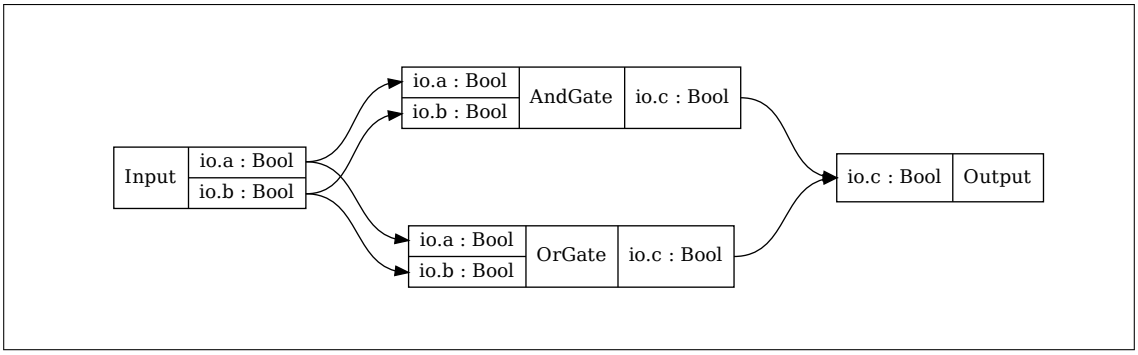

Figure 4.1: The graph we will use in order to compare several viewing libraries. This model includes two logical components : a AND and a OR gate and two nodes which are representing the input and output of the parent component.

4.1 GraphStream

GraphStream is a Java library used for the modeling and analysis of dynamic graphs[3]. The goal of the library is to provide a way to represent 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 shows the base graph model realised with the GraphStream library.

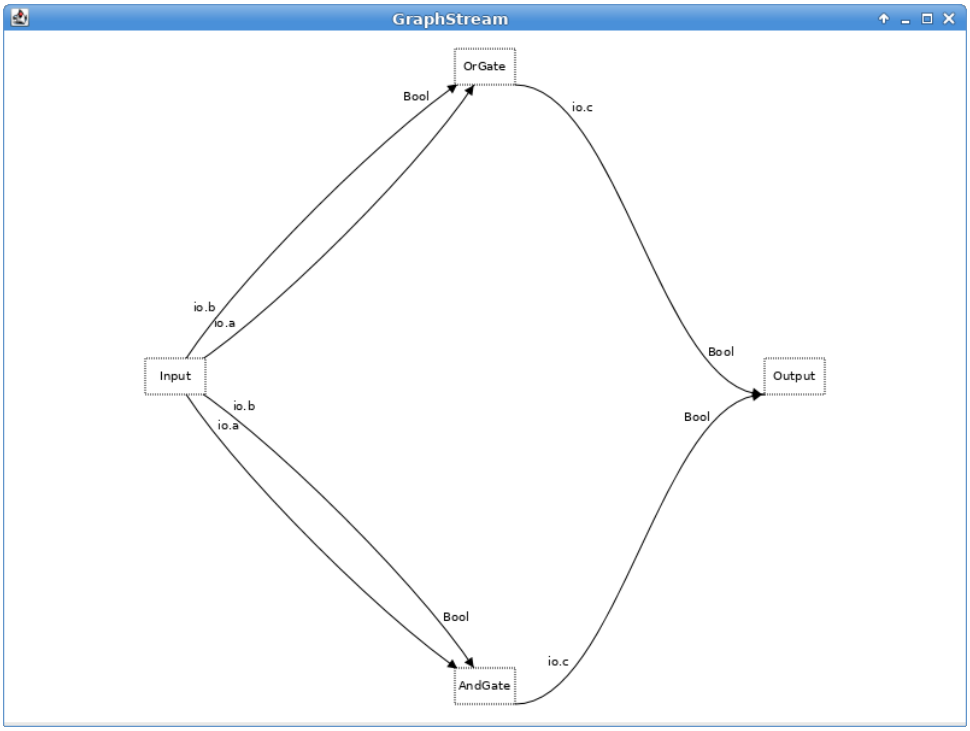

Figure 4.2: The base graph model rendering using GraphStream

GraphStream allows us to simply create a graph (or a multigraph) in Java like in listing 4.1.

```
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 4.1: Modeling 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 interesting for hardware component visualization : 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 additional special case is the label on the edges. With GraphStream, we could add some labels on the edges like in listings 4.2, but we can't add multiple ones. To add multiple labels we need to use sprites like in listing 4.3.

```
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 4.2: A complete example on how to add a label to an edge with the GraphStream library.

```
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 4.3: A complete example on how to add multiples labels to an edge with the GraphStream library. This time, we have to use sprites.

4.1.3 Conclusion

GraphStream owns other features which could be useful 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 that we need to indicate the component information using label on the edges. The figure 4.2 shows four nodes and six connections, but if we have a lot more edges between nodes, it would be difficult to see the type of the inputs and outputs, as shown in figure 4.3.

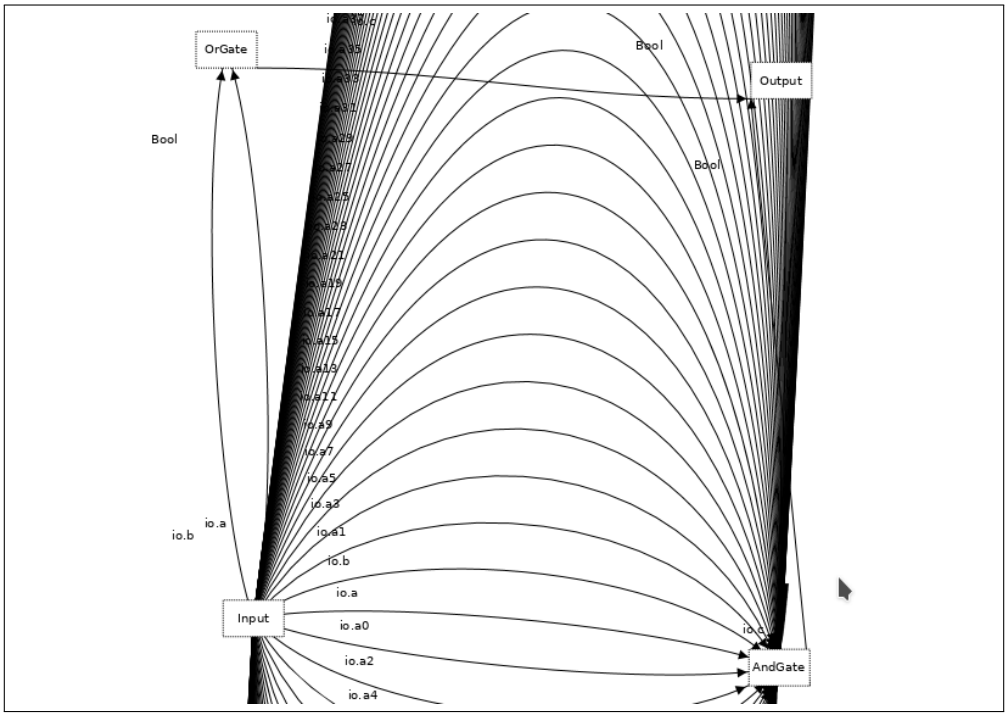

Figure 4.3

In order to overpass those problems, we need to visualize the type in another form : in the graph 4.1 we had the idea to show the inputs and outputs as a port of the component. With GraphStream we can't do it, so we need another library which has this idea of port.

4.2 Draw2D

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

- 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 indicate that the GraphStream library is lacking the idea of port, but Draw2d already has them.

The listing 4.4 shows 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.

The code in the listing 4.4 is producing the HTML page shown in figure 4.4.

```
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 4.4: 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.

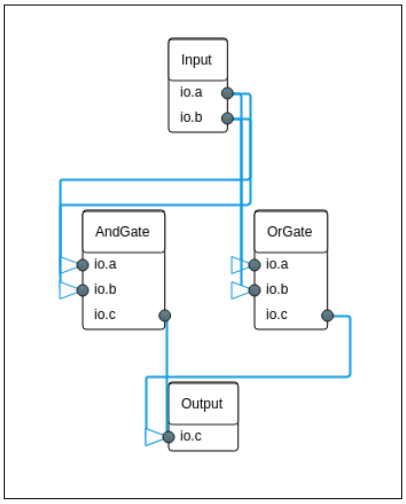

Figure 4.4: Rendering of the base graph model diagram from figure 4.1 using the Draw2D library

4.3 Graph layout

A feature that Draw2D doens't own is the layout of the diagram. If we did not say anything about the position of the node, Draw2D simply overlaps all the nodes as illustrated in figure 4.5.

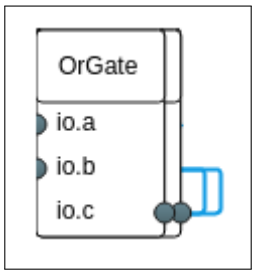

Figure 4.5: When we don't specify the nodes positions with Draw2D, the engine just overlaps all the nodes.

In order to produce a visualisable diagram, we need to layout ourselves the diagram. We need to mention that 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 own ports, so we need to layout the edges on a specific part of the graph.

The first two specialties aren't a big deal, but the last one is quite problematic. It involves a special layout algorithm which takes care of the ports position on the nodes. Otherwise the visual result could be chaotic like in figure 4.6, in which we can't see the connections between some specific ports.

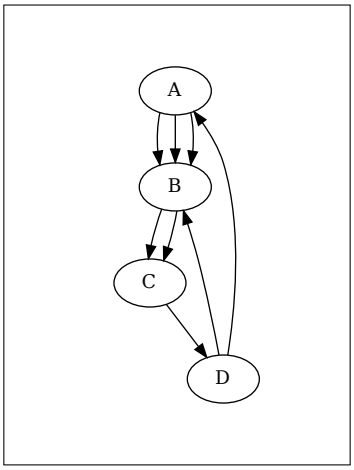

Figure 4.6: Visualization of a multi-graph without port. We can't differentiate the connections between two differents ports on the same component. It would be feasible if we indicate the signal name on the edges, but the problem from figure 4.3 still remains.

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

4.4 Conclusion

GraphStream looks like a really good library for manipulating graphs 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 looks ugly (figure 4.3). Then we have to find a library which allows this idea of ports. We found it with Draw2D and chose it for the project. The disadvantage of using this library is that we can't layout easily the graph by ourselves.

5 Parsing the AST

The abstract syntax tree of SpinalHDL is the core component of the whole KlugHDL product. It is used to produce the intermediate representation in the generation pipeline (figure 3.1) and once the intermediate representation is built, the rest is easily doable.

The concept of AST has already been explained 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 separated in multiple phases :

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

As illustrated in the intermediate model class diagram 3.6, the intermediate representation is composed as follows:

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

A model could be represented, using Scala, with the class declaration in listing 5.1. 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 5.1: 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 has no parent.

5.1 Diagram parsing

Generating a diagram (see listing 5.2) is trivial because we just have to retrieve all the components which are parents, the algorithm is implemented in listing 5.3.

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

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

Listing 5.2: 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 5.3: This function parses the AST and generates all the corresponding diagrams objects for a specific component

Because we are using a set, we don't have to check if the diagram already exist. To make this possible we have to redefine the `equals` function like in listing 5.4.

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

Listing 5.4: We have to override the equals function in order to use the diagram 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 5.6 and the KlugHDLComponent is declared in listing 5.5.

```
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 5.5: Declaration of the components class (here named KlugHDLComponents). There are two types of components : the basic ones and the ones that represent the external world input and output as in figure 3.5.

```
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 5.6: Components parsing and generation in Scala for one diagram

Note that we generate the interface for the external world connections in the model. These 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 Component class already owns a `getAllIo` function which returns all the inputs and outputs nodes for a component. So we just need to create a port object (listing 5.7) from those and add it to the correct KlugHDLComponents. The algorithm is implemented in the listing 5.8.

```
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 5.7: Declaration of the Port class with Scala. There are two types of ports : input and output.

```
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 5.8: Ports parsing and generation in Scala for one diagram, the ports are generated component by component.

5.4 Connections parsing

The connections parsing is the hardest implementation to do because we have multiple types of connections :

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

In order to find the starting or the ending point of a connection, we have to cross all the AST until we find all the inputs or outputs.

5.4.1 Input connections

The input connections represent all the connections generated from the inputs of a node. Firstly, we have to retrieve all the inputs for a specific AST node then we need to generate a connection between two ports on two nodes. The implementation can be seen in listing 5.9.

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

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

    // iterate over all the inputs on each node of the AST until finding 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 5.9: Implementation in Scala of the parsing and generation of 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 5.10.

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

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

    // iterate over all the inputs on each node of the AST until finding 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 5.10: Implementation in Scala of the parsing and generation of all the input connections of 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 ones. Because we just have to generate the inputs and consumers and subtract them like in a set. The algorithm is described in listing 5.11.

```scala
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 5.11: Implementation in Scala of the parsing and generation of all the connections (inputs 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 cases 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 used by KlugHDL in order to produce an output for further use. Then we will see the static and dynamic visualizations which are available with KlugHDL.

6.1 Intermediate representation

The intermediate representation generated 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 outputs are called "backend".

6.1.1 The DOT backend

DOT and Graphviz are two software for diagram visualization. The diagram is written using the DOT language and then compiled to various image formats like png or pdf. In addition to the image generation, DOT and Graphviz also hold a very powerfull layout engine which can be useful for future implementation.

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

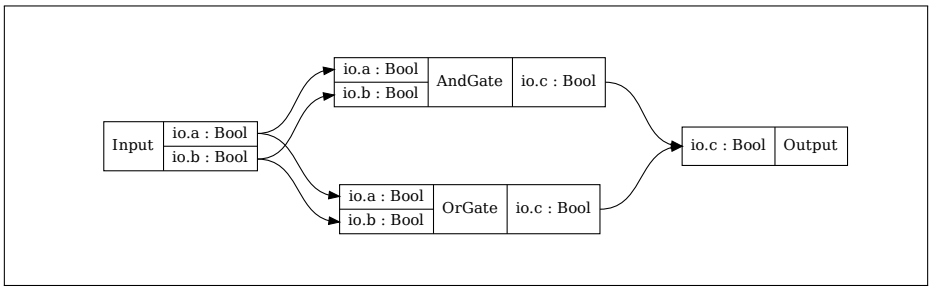

Figure 6.1: A complete Graphviz diagram generated by DOT, the code of this example is available in listing 6.2

```
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 generate the ports on the nodes and link them with the connections.

6.1.2 The JSON backend

JSON (JavaScript Object Notation) is a format mostly used by Javascript for object serialization. The format is completely included in the Javascript language which is able to directly understand the object notation without any additional code to write.

At the beginning of the project, we tried to produce the Javascript output directly from the intermediate model. The problem occurs when we change the library we are using, we need 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 used the Lift-JSON library [6]. This library is very easy to use and offers a small DSL in order to write readable code. The listing 6.1 shows 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 6.1: The lift JSON library offers the opportunity to write readable and scalable code with her DSL

6.2 Static visualization

With KlugHDL it's very simple to generate a diagram in a pdf format. For example, this could be useful for a report documentation. The listing 6.2 shows 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 6.2: 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 provided with KlugHDL. The listing 6.3 illustrates 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 6.3: A complete KlugHDL example on how to generate a JSON model

6.3.1 Extending the draw2d library

We have to extend the draw2d library in order to create the specific diagram we want. The only addition to make is to create another node for draw2d. This is done by extending the `VerticalLayout` shape. The whole new shape is created with the code in listing 6.4.

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

Currently, 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 components are, on the double click input, calling the callback method multiple times. This creates, first, a very long callback and then corrupts the memory and raises exceptions.

In order to find a solution to this problem, we have to re-design the JavaScript implementation of the dynamic visualization and also the JSON format generated by KlugHDL.

Thanks to the JSON model, we can also create a dynamic visualization with another library than Draw2D or with another language than JavaScript.

6.4 Current working state

KlugHDL is able to parse and generate the intermediate model for simple SpinalHDL component like the ones implemented in listing 3.1. The static visualization, generated by DOT and Graphviz, is working too. The dynamic visualization is generated and displayed in a web page. The navigation to a child from the current components is also working but the opposite is problematic as explained in section 6.3.2.

```
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 6.4: 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

6.5 Further work

We can do a lot more to complete and improve KlugHDL :

- Parsing and generating the intermediate model for any kind of SpinalHDL component
- Finding a way or improve the library to directly manipulate the Draw2D layout
- Generate the layout information with DOT and Graphviz and add them to KlugHDL

All the work is available at the following git repository : `https://github.com/SnipyJulmy/MSE_1617_PA` and under the GNU public license version 2.0.

7 Conclusion

This project tries to offer an additional tool for SpinalHDL in order to increase its popularity. Domain specific languages are more and more used in computer software nowadays and KlugHDL could be described as a tool for development and documentation.

Writing VHDL code over and over again seems to be painful and SpinalHDL tries to make the developer life easier, but VHDL owns a lot of tools to increase productivity. KlugHDL is the first "tool" of the whole world of SpinalHDL and launches a new era for SpinalHDL and its believers.

KlugHDL is able to parse and generate static diagrams with DOT and Graphviz for any simple SpinalHDL component. Components with bus or complex basetype, like $Uint(nbits)$ aren't supported for the moment. KlugHDL is also here to demonstrate that it is possible to generate diagrams from the SpinalHDL AST.

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

There is a lot of work to do on KlugHDL in order to have a complete tool for diagrams generation. It is not only about the visualization but also about the kind of SpinalHDL components to parse.

Some other features and work can be added to KlugHDL :

- Integrate the diagram visualization to an integrated software development like Eclipse or Intellij for a live preview of the current work.
- Generate the diagram without generating the whole RTL for a component.

I would like to sincerely thank the following people, whitout whom this project would have been impossible to realize :

- Professor Dr. Mudry Pierre-André, for coordinating and supporting me
- Charles Papon, author of SpinalHDL, for answered my questions about SpinalHDL and supporting me
- Roland Julmy, my father, for having review this document and encouraging me
- Marcel Julmy, my uncle, for having review this document.

Fribourg, the 17th January 2017

Sylvain Julmy

Sworn declaration

“ I hereby solemnly declare that I have personally and independently prepared this paper. All quotations in the text have been marked as such, and the paper or considerable parts of it have not previously been subject to any examination or assessment.”

List of Figures

List of Listings

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 wants 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 activates the compilation options

Standard progress :

1. The user indicates to the compiler to launch KlugHDL
2. The user compiles the program using the standard Scala compiler
3. A Windows appears and shows the component diagram

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

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

Postcondition : The component diagram is displayed

3 : HCI needing (Optional)

A windows that shows the component diagram

4 : Complementary remarks

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

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 wants to see the type of some edges
Authors : Sylvain Julmy

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

2 : Sequence description

Precondition : The diagram is generated and opened.

Standard progress :

1. The user clicks on "Show types"
2. The diagram displays the types of the edges

Postcondition : The types of the selected edges are displayed.

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 wants 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 opened.

Standard progress :

1. The User double clicks on a component
2. The system displays the inside of the component

Postcondition : The double clicked component is maximized 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 for the leaf component which has no sub-component and the BlackBox, every component is zoomable.

D Facts sheet : Exit a component

Exit a component

1 : Identification summary

Abstract : The user wants to exit (zoom out) a 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 user has entered a component previously.

Standard progress :

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

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

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

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

3 : HCI needing (Optional)

- A visualization of the component and its parent

4 : Complementary remarks

Except for 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 wants to display and view the component diagram of its 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 entered the correct option for the compilation.

Standard progress :

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

Postcondition : The diagram is displayed.

3 : HCI needing (Optional)

- A diagram of the component of the user program

4 : Complementary remarks

The diagram could be generated but not necessarily displayed 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 wants 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 opened.

Standard progress :

1. The user clicks on the "Hierarchical tree view" button
2. The hierarchical tree view is displayed or hidden

Postcondition : The hierarchical view is visible if it was previously hidden and is not visible if it was previously displayed.

3 : HCI needing

- A tree view of the componenent and its 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 wants to generate the RTL of its 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 compiles 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 wants 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 opened.

Standard progress :

1. The user enters the text specific to a component (class name)
2. The view turns gray the component that are not matching the text

Postcondition : The elements that are not matching the introduced pattern are grayscale.

3 : HCI needing (Optional)

- A writable textfield

4 : Complementary remarks