SyncSim Extensions
Zooming, Routing and Editing

Petrus Bergman

Luleå University of Technology
MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of EISLAB
SyncSim Extensions

Zooming, Routing and Editing

Petrus Bergman
epeeba-0@student.ltu.se

January 24, 2008
Preface

This thesis is the result of my final years of the Master of Science in Computer Science and Engineering program at Luleå University of Technology (LTU). The work was done at the Department of Computer Science and Electrical Engineering (CSEE) at LTU.

In the spring term of 2005, I attended a course in Computer Architecture where I and a few other students were to make an Intel 8086 model implementation for simulation in SyncSim. That was my first experience with SyncSim and I quickly noticed that it was rather complex to work with as a designer/developer. The idea for this thesis started to take form during that course and the work with my SyncSim extensions began late summer 2005. The workload became somewhat larger than first expected and due to that and some other circumstances, the thesis took a lot longer to finish than supposed to. The practical task (code implementation) was finished late summer 2006, while this report was not finished until late 2007.

I would like to thank Per Lindgren for his role as supervisor of this thesis and for him patiently waiting for this thesis to be finished. I would also like to thank Andrey Kruglyak for his support.

Petrus Bergman

November 2007
Abstract

SyncSim is a Java based simulator framework capable of simulating hardware models described with Java or VHDL, depending on the used simulator module. SyncSim is used for educational purposes together with a model of a MIPS processor core in courses given at Luleå University of Technology.

The purpose of this thesis is to extend SyncSim with a zooming system, an automatic router system and some design editing tools. Zooming and panning will facilitate use of SyncSim when teachers need to focus on a specific part of a design during a lecture. Automatic routing of design components and access to editing tools will greatly reduce the development time of new designs or when making design modifications.

The result is a new version of SyncSim with enhanced support for teachers and designers/developers. An automatic routed MIPS design will have a similar bus layout as a manually created MIPS design. Designs can be modified graphically within SyncSim instead of writing XML code.
# Abbreviation list

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWT</td>
<td>Abstract Window Toolkit</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versions System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HCG</td>
<td>Horizontal constraint graph</td>
</tr>
<tr>
<td>LEA</td>
<td>Left Edge Algorithm</td>
</tr>
<tr>
<td>MIPS</td>
<td>RISC microprocessor architecture developed by MIPS Technologies</td>
</tr>
<tr>
<td>NP</td>
<td>Non-deterministic Polynomial time</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>VCG</td>
<td>Vertical constraint graph</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuits</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
# Contents

1 Introduction ................................................................. 1  
   1.1 Background................................................................... 1  
   1.2 Purpose ...................................................................... 1  
   1.3 Limitations .................................................................. 2  

2 Zooming ........................................................................ 3  
   2.1 Background .................................................................. 3  
      2.1.1 SyncSim example design ...................................... 3  
      2.1.2 The design XML file ............................................. 4  
      2.1.3 The graphical component ..................................... 4  
      2.1.4 SSComponent ..................................................... 5  
   2.2 Implementation .......................................................... 5  
      2.2.1 A workspace panel with zoom and pan support ....... 5  
      2.2.2 ZComponent ....................................................... 7  
      2.2.3 SyncSim modifications .......................................... 9  
      2.2.4 Design specific modifications ............................... 9  
   2.3 Evaluation ................................................................... 10  
      2.3.1 Result .................................................................. 10  
      2.3.2 Zooming and panning shortcomings ....................... 10  
      2.3.3 SyncSim 2.1 ......................................................... 10  

3 Routing ........................................................................... 11  
   3.1 Background .................................................................. 11  
      3.1.1 Wire routing ........................................................ 11  
      3.1.2 Global and local routing ....................................... 12  
      3.1.3 Graph models ..................................................... 13  
      3.1.4 Spanning trees & Steiner trees .............................. 14  
      3.1.5 Maze routing ....................................................... 15  
      3.1.6 Net ordering ........................................................ 17  
      3.1.7 Channel routing ................................................... 19  
      3.1.8 Switchbox routing ............................................... 22  
      3.1.9 Nets in SyncSim ................................................... 22  
   3.2 Implementation ........................................................... 24  
      3.2.1 Prerequisites ........................................................ 24  
      3.2.2 Providing the floorplan ......................................... 24  
      3.2.3 Providing the net list ............................................. 24  
      3.2.4 SSBus ................................................................. 25  
      3.2.5 Interfacing with SyncSim ...................................... 26  
      3.2.6 Router configurations .......................................... 27  
      3.2.7 Preparing to route ............................................... 28
3.2.8 Global routing ................................................................. 29
3.2.9 Detailed routing .............................................................. 33
3.2.10 Presenting the routing solution to SyncSim.............................. 40
3.2.11 Design XML file modifications ........................................... 41
3.2.12 SyncSim modifications ..................................................... 44
3.3 Evaluation ........................................................................... 45
  3.3.1 Router shortcomings ....................................................... 45
  3.3.2 Simulator speed ............................................................ 45
  3.3.3 The design XML file ....................................................... 45
  3.3.4 SyncSim 3.0 .................................................................. 45
  3.3.5 Test case – a routed MIPS design ...................................... 46

4 Editing ................................................................................. 49
  4.1 Background ......................................................................... 49
    4.1.1 The design XML file ....................................................... 49
    4.1.2 Modifying existing designs ........................................... 50
    4.1.3 Problem description ..................................................... 50
  4.2 Implementation .................................................................. 50
    4.2.1 Overview ...................................................................... 50
    4.2.2 The XMLWizard ........................................................ 50
    4.2.3 Edit handler ............................................................... 52
    4.2.4 Selecting components to edit ........................................ 53
    4.2.5 Moving components .................................................... 54
    4.2.6 Resizing components ................................................... 54
    4.2.7 Moving ports ............................................................. 54
    4.2.8 Object snap ............................................................... 55
    4.2.9 Edit listeners .............................................................. 56
    4.2.10 SyncSim modifications ............................................... 56
  4.3 Evaluation ......................................................................... 57
    4.3.1 Result ......................................................................... 57
    4.3.2 Editing shortcomings ................................................... 57
    4.3.3 Design development time ............................................. 57

5 Conclusions ........................................................................ 59

Bibliography .......................................................................... 61
Chapter 1

Introduction

1.1 Background

In spring term 2004, Erik Byström, Lars Magnusson and Robert Selberg developed a general synchronous circuit simulator during the course ‘Project in Digital Synthesis’ at Luleå University of Technology. The simulator, called SyncSim [1], is a Java based simulator framework that let the user load and simulate different hardware models. It has the ability to use different simulators as well, which makes it very flexible and versatile. Because it is written entirely in Java, it is also platform independent and can be run on any Java supported system. In recent years, SyncSim has been further developed and extended by Jimmie Wiklander, Andrey Kruglyak, Jan Enoksson and Simon Olsson.

A hardware design that is loaded into SyncSim consists of three parts; the functionality description of simulated components, the graphical representation of the components (optional) and an XML file describing which simulator to use and how the simulated components are connected and how they are laid out visually. The functionality description may be written as Java class files or VHDL files [2] depending on what simulator that is used.

SyncSim provides a Graphical User Interface (GUI) that lets the user to view and control the simulation. The user can step forward, step backward, free-run and pause a simulation. The design’s graphical representation may allow the user to view the state of each simulated component at the current clock cycle of the simulation.

At Luleå University of Technology, SyncSim is used for educational purposes in courses given by Embedded Internet Systems Laboratory (EISLAB). For example, in the course ‘Computer Organization and Logic Design’, SyncSim gives the students the opportunity to study and simulate the core of a simplified MIPS processor.

1.2 Purpose

Because SyncSim is used in educational purposes it should not only be easy to use for the students, but it also needs to be practical for the teacher of the class. This means e.g. that the teacher should be able to use a video projector to show designs loaded into SyncSim and to use the simulator to show the data flow through the design. Up to now, this has not been very practicable for large designs. What is needed is the ability to zoom in to a region of the design so it will be clearly visible for the audience. Thus, the first part of this thesis is about to develop and implement a system for zooming ability.

As to date, creating a new design for simulation in SyncSim is not a trivial task. One of the most laborious parts is to write the XML file that describes the layout of the design. The designer must manually decide the location of each component that has a graphical
representation and must write several lines of XML tags to describe exactly how a single wire between two components should be drawn. As most designs will have many wires, writing the XML code for them all is a huge task. If the designer later on needs to move some components, this also requires that some (or all) of the wire descriptions has to be rewritten. What is needed here are tools for layout-level design automation which will help the developer to create a design for simulation in SyncSim, and this is what the second, and main part of this thesis is about. The second part will consist of two intermediate goals; to implement a system for automatic routing and to develop tools for editing design components.

1.3 Limitations

Automatic placement and routing of components is not an easy task and has been studied a lot in the literature. Several useful algorithms exist. However, most algorithms are aimed at VLSI design automation where small distance between components are an important factor, as well as minimal routing of wires, utilizing the most of a chip’s different levels and making the design both cost effective and heat tolerant.

In the case of the SyncSim software and its aim at teaching and learning to understand synchronous/combinatorial circuits, it is not important trying to optimize production costs and dealing with heat problems by using some smart and complex automatic placement algorithms. Considering that, it has been decided to let the user be as free and creative as possible when it comes to layout of components and thus no automatic layout algorithms are used. The user will place the components manually by moving/resizing each component at will. This should help the designs to be more simple and easy to understand rather than optimized and hard to take in.

Opposite to placement of components, wire routing is considered as one thing that can benefit from being automated. Thus, an automated wire router will be developed. The focus will be on creating a router with support for several options aimed at simplifying routed wires. One of the main goals of the router will be to generate wires that are easy to follow.

The different modules for zooming, routing and editing should be as self-contained as possible to make them useful in any application that can benefit from zooming capabilities and/or automatic routing. Still they must be well integrated with SyncSim and general enough to work with any design. To keep the platform independence of SyncSim, all extensions to it will be implemented in Java.
Chapter 2

Zooming

2.1 Background

This section will describe how SyncSim represents graphical design components and prepare the reader to understand the implementation description of the zooming extension module in section 2.2. The focus here is only the parts that are relevant to understand the next section.

2.1.1 SyncSim example design

Figure 2.1.1 shows the graphical representation of a simple example design loaded into SyncSim 2.0. The example design shown here is the design that will be used as an example throughout the rest of this report.

The small black boxes to the left and to the right are synchronous registers that forwards the signal value from the input wire to the output wire each clock cycle.

The box with a ‘+1’ is an adder that will add 1 to its input or subtract 1 depending on the counting direction decided by the box with a ‘dir: +‘. When the output from the adder reaches 10, the direction changes, and the adder will start to subtract 1 from its input signal. When the value reaches 0, it will start counting up again.

The bottom box colored red is the monitor component that displays the current value from the value register.

Each component, including the wires, has a tool tip text with the component name and a list of the in and out ports and corresponding port signal value. Thus, it is possible to look at the signal value in any part of the design at each clock cycle during simulation.

Figure 2.1.1: An example design loaded into SyncSim.
2.1.2 The design XML file

SyncSim uses an XML file to describe the graphical composition and layout of a design. The following XML code snippet is from the example design described in section 2.1.1.

```xml
<!-- Compare -->
<component name="Compare" class="example.Example_ComparePart">
   <gfx class="example.Example_CompareComponent" width="100" height="110" x="300" y="20">
      <property bitWidth="4" />
      <property topValue="10" />
   </gfx>
</component>
```

The interesting part here is the ‘gfx’ tag that tells SyncSim to load the Java class ‘example.Example_CompareComponent’ as the graphical representation of the component named ‘Compare’. The width, height, x and y attributes sets the location and size of the graphical component.

2.1.3 The graphical component

The code snippet in section 2.1.2 refer to ‘example.Example_CompareComponent’ as its graphical representation. If we have a look at the file Example_CompareComponent.java, it contains the following (non-relevant parts replaced with ‘...’):

```java
import java.awt.Graphics;

public class Example_CompareComponent extends SSComponent {  
   
   public void paint(Graphics g) {
      int height = getHeight();
      int w = getWidth() - 1;
      int h = height - 1;
      g.drawRect(0, 0, w, h);
      ...
      g.drawString(dir, 2, (h + fontSize * 12 / 19) / 2);
      ...
   }
   
   public void update() {
      repaint();
   }
}
```

This Java class file extends the SSComponent (see section 2.1.4) and contains a paint method that draws the content of the component; in this case a black border and a text string representing the current direction (‘dir: +’ or ‘dir: -’). The width and height of the component is provided by the getWidth and getHeight methods.

The update method is called by the simulator each clock cycle and results in a repaint of the component.
2.1.4 SSComponent

The SSComponent is part of the SyncSim framework and is the base class for all graphical components in a SyncSim design. SSComponent provides design components with multi-line tool tip that displays all inputs and outputs as default. It also provides a system for popup menus if needed. Because all graphical design components extend SSComponent, SyncSim knows how to access them as they all have a common interface. SSComponent also acts as a standard Java graphical component as it extends JComponent from the Swing widget toolkit.

The relationship between the XML file, the graphical representation of the design component and SSComponent is illustrated in Figure 2.1.2 below.

![Figure 2.1.2: The relationship between the XML file, the graphical representation of the design component, SSComponent and JComponent](image)

2.2 Implementation

In this section, the implementation of the zooming extension module in SyncSim will be described and explained. No actual code will be presented other than small code snippets of some of the changes that has been made to existing class files. The interested reader is referred to the CVS repository [3] containing both the source for SyncSim as well as the zooming module. Javadoc documentation can be generated from the source code and may be consulted during perusal of this section. The zooming module was implemented using SyncSim 2.0 as a code base. All referred class names starting with Z, Zoom or Pan are new class files that are part of the zooming module. Apart from SSComponent, all other referred classes are part of Java’s runtime environment.

2.2.1 A workspace panel with zoom and pan support

SyncSim’s original workspace panel is set to a fixed size when a design is loaded. That is not optimal for a zooming module as when designs are magnified, the size of the workspace panel should scale accordingly. Thus, the workspace panel was replaced with a ZCanvas. The ZCanvas extends a JPanel from the Swing widget toolkit and adds support for zoomable and panable components. It has dynamic bounds and updates itself so that it always has room for all components it contains.
**Zoom handler**

The `ZoomHandler` class acts like a ‘brain’ to the zooming system and decides how and when zooming should be executed. One ZoomHandler handles one ZCanvas and adds listeners to the ZCanvas in order to respond to user events.

The ZoomHandler maintains a list of possible zoom levels that is used to decide the next zoom level when zooming in or zooming out. The zoom level list is not fixed. New zoom levels can easily be added to or removed from the zoom level list during runtime.

During ZCanvas initiation, the ZoomHandler is created and added on top of the ZCanvas container to act as a display component for current zoom level status. The zoom level status is shown whenever the zoom level changes. Controlled by a timer, the status is only visible for a few seconds. See **Figure 2.2.1** below for an example.

**Zooming**

Zooming can be initiated by either pressing the ‘Zoom in’ or the ‘Zoom out’ icons in the SyncSim toolbar, or by using the key shortcuts ‘+’ or ‘-’ on the keyboard.

![Figure 2.2.1: Zooming in to the next zoom level. The zoom level status is visible in the upper right corner during a few seconds after each zoom change.](image)

The ZCanvas makes sure all zoomable components are zoomed by calling the `zoom` method on each component that implements the `Zoomable` interface. ZCanvas then updates the canvas bounds, pans components if necessary and adjusts the workspace scrollbars to make sure the correct workspace region is within the visible area. Finally, it broadcasts a `ZoomEvent` to all registered `ZoomListeners`. If components have been panned, a `PanEvent` is broadcasted to all `PanListeners`.

As default, the center of the zoom is the center of the visible workspace area. However, zooming is supported around any point. By using the ‘zoom select’ tool from the toolbar, the user can select and zoom into any region on the workspace. See **Figure 2.2.2** below. The region zoom is not limited to the list of available zoom levels. Instead, a new zoom level is calculated and temporarily added to the zoom level list. That temporary zoom level is then removed next time the zoom level is changed.
Panning

Panning is initiated by pressing left mouse button on the workspace area and dragging it in any direction. The ZCanvas grows its bounds as necessary when any component reaches and tries to pass the bounds during panning. This means that the mouse event coordinates that initiated the panning may require to be translated as the origin depends on the canvas bounds. ZCanvas takes care of all coordinate translations, updates the canvas bounds, and moves panable components by calling the pan method on each component that implements the Panable interface. Finally, it broadcasts a PanEvent to all registered PanListeners.

2.2.2 ZComponent

To utilize all features of the ZCanvas, graphical design components added to it must implement both the Zoomable and Panable interfaces. As all graphical design components already extend a common SyncSim base class, the SSComponent, the obvious choice was to make the SSComponent take care of the implementation. Actually, there were two possible options; incorporate all zooming and panning code within the SSComponent class, or let SSComponent extend another class that will handle zooming and panning, and in turn be the connection to a JComponent. To keep the zooming module as independent from SyncSim core files as possible, it was decided that the second option would be the best way to go.

Component bounds

The bounds of a JComponent define the location and size of a component and thus the available painting area. Anything outside these bounds is invisible.

Zooming a component means that the contents of the component is enlarged or reduced. The painting area must be rescaled as well, which means that the component bounds must be modified. To avoid rounding errors when zooming in and out, the ZComponent keeps track of the component’s original 1:1 bounds and rescales the component relative to these bounds according to the current zoom level.

The ZComponent makes sure that a change of the component bounds in the 1:1 coordinate system is reflected on the component bounds in the current coordinate system, and vice versa.
Painting the component

Zooming is not just about resizing the component bounds. For a real zooming effect, all contents that are painted must be magnified. For example, a 1 pixel thick line should be drawn as a 2 pixel thick line, if the zoom level is 200%.

The code snippet in section 2.1.3 shows that graphical design components use the `paint` method to draw their contents. Current component bounds are used as bounds for the available painting area. However, this method will not work well with zooming. The problem is that ZComponent resizes the component bounds when changing the zoom level, but that only results in a larger (or smaller) painting area. A 1 pixel thick line will still be drawn as a 1 pixel thick line, only the length will change. Thus, there will be no real zoom effect. The solution would be to translate the `Graphics` object into a `Graphics2D` object and apply an `AffineTransform` to rescale all graphics.

The decision whether the solution should be implemented in the design class file, or in the ZComponent base class, fell on the ZComponent. There are not any advantages of implementing it in the design class files other than doing so it would be possible to keep using the standard `paint` method in these classes. On the other hand, implementing it in ZComponent, gives the advantage that it is completely transparent to the designer and it avoids duplicated code, but it will introduce a new paint method.

When Java’s graphics engine calls the `paint` method to paint a design component, the `paint` method in ZComponent applies a transform on the graphics object and forwards the call to the design-specific component together with the rescaled graphics object and the size of the available painting area as parameters.

Instead of calling `public void paint(Graphics g)` in the graphical design component, the ZComponent calls the following method:

```java
public void paint(Graphics g, int width, int height)
```

If the code snippet from 2.1.3 were modified to use the new paint method, it would look like this:

```java
... public class Example_CompareComponent extends SSComponent {
  ... public void paint(Graphics g, int width, int height) {
    int w = width - 1;
    int h = height - 1;
    g.drawRect(0, 0, w, h);
    ... g.drawString(dir, 2, (h + fontSize * 12 / 19) / 2); ...
  } 
  ...
  public void update() {
    repaint();
  }
}
```

Note that as the painting area size is provided as parameters, it is no longer necessary to call the `getWidth` or `getHeight` methods. In fact, using these methods is forbidden, as they will return wrong bounds, i.e. current component bounds instead of 1:1 component bounds.
2.2.3 SyncSim modifications

One of the objectives of the zooming module was to make it as self-contained as possible so it can be used in any applications that might benefit from zooming tools. However, some changes had to be done in the SyncSim core files in order to incorporate the zooming module in SyncSim.

The zooming module was implemented using SyncSim 2.0 as a code base. Below is a list of existing SyncSim class files that were affected by the implementation of the zooming extension module. For each class, a short description tells which functionality and/or responsibility that have been added to these files.

syncsim.SSComponent
- Extends ZComponent instead of JComponent to support zoom and pan
- Implements the new paint method to draw a default outline box

syncsim.SyncSimFrame
- Uses a ZCanvas as a workspace panel instead of a JPanel to support zoom and pan
- Adds quick menu toolbar buttons for zoom and pan functions
- Adds ZCanvas as a component listener to the workspace scroll pane
- Configures the workspace scroll pane to always show its scrollbars

2.2.4 Design specific modifications

All existing graphical design components have to be updated with a new paint method. See page 8 for details.
2.3 Evaluation

This section will discuss some strengths and weaknesses of the implemented zooming module as well as the value of the result in respect to the purpose of this thesis.

2.3.1 Result

The implementation of the zooming module has resulted in a general zooming and panning system that will work for any designs that uses the new paint method to draw the component graphics. In the beginning, the intention was to try implementing the zooming system without affecting current design implementations, but as the problem with the paint method became apparent (see page 8 for details), it was no longer possible to avoid modifying existing design files.

The zooming effect is ‘real’ in the sense that all graphics is magnified, lines will be thicker, texts will be larger etc. For educational purposes, teachers can easily use the zoom and pan features to focus on a specific part of a design while talking about that specific part on a lecture and it will be clearly visible for the audience.

2.3.2 Zooming and panning shortcomings

Zooming and panning seems to work flawlessly, but there are always things that can be better.

Tooltip zooming
The zooming system is currently only targeted at handling design component zooming, but it might be useful if design tooltips can be zoomed as well.

Scroll wheel support
Zooming actions is either available via the SyncSim toolbar or key shortcuts, but there is no support for zooming with the mouse scroll wheel, which might be useful for some users.

Fit to workspace
There is currently a toolbar button and a key shortcut for restoring the zoom level to 100%. There is also a toolbar button and a key shortcut to center the design on the workspace area. However, there is no simple action to fit the entire design on the workspace.

2.3.3 SyncSim 2.1

The code base for this thesis was SyncSim 2.0. The zooming extension module was the first part of this thesis to be completed and resulted in a release of SyncSim 2.1. In order to get current MIPS designs compatible with the zooming system, all graphical component class files were updated with the new paint method required by SyncSim 2.1.
Chapter 3

Routing

3.1 Background

This section will introduce concepts and the theory behind basic wire routing and prepare the reader to understand the implementation description of the automatic routing module in section 3.2. As this thesis is limited to the design of a router aimed at simplifying routed wires and to generate wires that are easy to follow, the focus will be on the general routing problem of finding a path between a source and a destination. Cost effectiveness and electrical issues will not be discussed although they are very important in real world implementations.

3.1.1 Wire routing

The object of routing is to find a set of connections that will wire together the terminals of different modules or blocks on a printed circuit board or computer chip. Each connection or net connects a source terminal to one or several destination terminals. The primary tasks of the router is to create geometries such that all terminals assigned to the same net are connected, to make sure that no terminals of different nets are connected and to obey all design rules. A router can fail by not connecting all terminals within a net (resulting in an open net), by mistakenly connecting two nets (short circuit) or by violating some design rule. In addition to correctly connecting the terminals, routers may also be expected to make sure the design meets timing, has no crosstalk problems, meets any metal density requirements, and so on.

Humans have traditionally outperformed computer routers by far, as early routers where not that sophisticated, but as designs have become larger and more complex, computer routers are in these days required in most cases. Several router algorithms have been developed over the years and computer routing tools have constantly improved.

The standard type of router for arbitrarily placed blocks is the maze router [4]. The maze router is guaranteed to find the shortest path for a single net between two terminals if it exists. However, the problem begins when more than two terminals have to be connected. The optimum path through these is the Steiner tree [7], which is an NP-complete problem, i.e. there is not yet any efficient method to solve it optimally. Furthermore, if there is more than one net to be connected, the solution will depend in most cases on the order in which the nets are routed. Finding the optimal order and hence the optimal solution becomes a very difficult problem. Add to this the long list of often-conflicting objectives as well as all electrical and technological constraints that must be applied and the routing problem gets even more complex.
3.1.2 Global and local routing

One strategy to solve wire routing is to split the problem into global (loose) wiring and local (detailed) wiring [6]. Global routing is the preliminary step and provides a plan on where to route every net. The actual connections and net locations is then determined during local routing.

Global routing

The input to the global router is a floorplan that includes the location and placements of all blocks in the design. The objectives of global routing may be one or more of:

- minimizing the total interconnect length
- maximizing the probability that the local router can complete the routing
- minimizing the critical path delay

Blocks in a design are in general of irregular shape and size. Thus, one important preparation step in global routing is to divide the routing region into more manageable pieces. The routing area can be divided into rectangular areas as shown in the example figure below.

![Routing channels and switch boxes for a placement of four blocks.](image)

Figure 3.1.1: Routing channels and switch boxes for a placement of four blocks.

A two-sided channel is a rectangular routing area with no obstructions inside, with terminals on two parallel sides and with wires entering and leaving through the other two sides. A switch box is a rectangular routing area with no obstructions inside and with wires entering and leaving in any direction. In Figure 3.1.1 the blocks in the design are illustrated as grey areas. There are six vertical channel regions labeled VC and seven horizontal channel regions labeled HC. The nine switch boxes labeled SB occurs where channels intersect.

The maximum number of wires allowed in one channel is defined as the channel capacity. Global wiring must ensure that all nets are routed without exceeding the capabilities of individual channels.

A common approach is to route one net at a time, choosing the shortest possible path. The problem can then be represented by a routing graph that depicts relationships between routing regions and nets to be connected. However, it is usually impossible to route all the nets with shortest connection paths because they all compete for the same space. Nets can be routed either independently or sequentially. Independently routing requires nets that cause congestion to be rerouted. When routing sequentially, the information about routing regions must be updated after completing each net.
Local Routing
After wire routing has been planned by the global router, the detailed (local) router operates on small regions of the floorplan to determine the exact location of each net. For example, Channel routers or switchbox routers can be used to perform this task. See section 3.1.7 and 3.1.8 for details.

3.1.3 Graph models
The global routing problem is usually translated into a graph model [7] that routing algorithms then can operate on. Two common graph models are the grid graph and the channel intersection graph.

Grid graph
In a grid graph, the routing area is divided into cells where each cell is represented by a vertex. Two vertices are joined by an edge if the corresponding cells are adjacent to each other. The occupied cells are represented as filled circles, whereas the others are represented as clear circles. See Figure 3.1.2 below.

![Figure 3.1.2](image1)

Figure 3.1.2: An example design divided into cells and its corresponding grid graph. The terminals a, b, c and d occupies one cell each.

Channel intersection graph
In a channel intersection graph the channels between the blocks are represented as edges and channel intersections are represented as vertices. The channel capacities are defined as edge weights. The terminals can also be represented as vertices. See Figure 3.1.3 below.

![Figure 3.1.3](image2)

Figure 3.1.3: An example design and its corresponding channel intersection graph.
3.1.4 Spanning trees & Steiner trees

Routing problems that has been translated into a graph model can be solved by finding a spanning tree of the graph.

A spanning tree of a graph \( G(V, E) \) is a tree \( T(V, E') \), where \( E' \subseteq E \), that connects all the vertices in \( G \). In Figure 3.1.4, each vertex represents a terminal and each edge represents a possible path between two terminals. Several algorithms exist that can find a spanning tree that connects all vertices. If each graph edge is labeled with the distance between the terminals to form a weighted graph, a minimum spanning tree can be found.

![Figure 3.1.4: The thick edges illustrate the spanning solution \( E' \).](image)

- a) A spanning tree \( E' = \{(a, c), (c, f), (f, g), (a, e), (e, h), (a, d)\} \)
- b) A minimum spanning tree \( E' = \{(a, c), (a, e), (e, f), (f, g), (e, h), (e, d)\} \)

For a multi-terminal net, it is easy to construct a minimum spanning tree to connect the terminals together, but the wire length will be unnecessarily large in most cases. To minimize the wire length, a Steiner tree is needed. A Steiner tree is a tree connecting all vertices in the graph as well as some additional nodes (Steiner nodes). When it comes to routing, usually rectilinear Steiner trees are used because wiring is in most cases restricted to be either horizontal or vertical. See Figure 3.1.5.

![Figure 3.1.5: a) A rectilinear minimum spanning tree. b) A rectilinear Steiner tree.](image)

Routing algorithms do their best to keep the wiring as short as possible. However, finding the minimum Steiner tree is an NP-complete problem, i.e. there is not yet any efficient method to solve it optimally.
3.1.5 Maze routing

Lee-Moore

One of the earliest routing schemes is the Lee-Moore maze routing algorithm [4][6][8], which will find the shortest path between any two points if it exists. This is done by first dividing the design area into a grid of cells and weighting each cell by its distance from the source of the wire to be routed. The distance is counted in a Manhattan manner, which means that paths will consist of only horizontal and vertical segments. The algorithm then enters an expansion phase where a breadth-first search is used to propagate a wave front from the source that will eventually reach the target if a connection is possible. A trace back phase then forms the connection by following any path with decreasing weight, i.e. decreasing distance from the source.

The strengths with the Lee-Moore maze algorithm is that it is guaranteed to find a path between two terminals for a given connection if such a path exists and that it is guaranteed to be a minimum path. However, the minimum path can only be guaranteed for one connection as when routing multiple connections, one connection may block other connections. The algorithm will require a vast amount of storage since large designs may contain millions of grid points. Another disadvantage is that the wave front extends in all directions, which makes the computation very slow. The time and space complexity for an M * N grid is $O(MN)$.

Reducing running time

There are some simple solutions to help reducing the running time of the Lee-Moore maze algorithm, as listed below and illustrated in Figure 3.1.7.

- Starting point selection: Choose the point furthest from the center of the grid as the starting point.
- Double fan-out: Propagate wave fronts from both the source and the target.
- Framing: Search inside a rectangle area 10-20% larger than the bounding box containing the source and target. If the search fails, it must be redone with a larger rectangle.

![Figure 3.1.6: The maze algorithm with a wave front propagating from the source. a) Partially completed expansion phase. b) Trace back phase. Routing completed.](image)
Weighted grid

In cases where other paths than the minimum distance path is desired it is possible to use a weighted grid. A weighted grid is a grid where each cell is weighted not by distance but by some other criteria. The solution will then be a minimum cost path. Consider the following example where each grid cell is weighted by the number of adjacent unblocked grid cells - 1:

**Figure 3.1.8:**

(a) The desired path is other than the minimum path.
(b) Initialize grid cell weights.

(a) Wave propagation. Accumulate grid cell weights.
(b) First wave reaches the target.

(a) Continue propagation to find other paths.
(b) Minimum cost path found.
Soukup’s algorithm

Soukup’s algorithm is one of many existing refinements of the maze router. It uses a combined breadth-first and depth-first search approach. A depth-first search is directed towards target T until an obstacle or T is reached. The depth-first search will keep change its direction so that it always points at T either horizontally or vertically. If an obstacle is reached a breadth-first search (as in Lee Moore) is used to “bubble” around it. The algorithm is illustrated in Figure 3.1.11.

![Figure 3.1.11: Soukup’s fast maze router algorithm. Filled circles = depth-first search, Clear circles = breath-first search.](image)

Soukup’s algorithm will find a path between the source and target terminal if it exists, but it may not be the shortest path. The time and space complexity for an M * N grid is still O(MN), but it is generally 10-50 times faster than Lee-Moore’s algorithm.

3.1.6 Net ordering

The order in which the nets are routed may greatly affect routing solutions. Consider the following simple example with two nets, a and b, on a single-layer routing area (nets may not cross over each other).

![Figure 3.1.12: Net ordering example.](image)
Figure 3.13 and Figure 3.14 illustrates two different scenarios. If net \textit{a} is routed before \textit{b}, the minimum path for \textit{a} has two solutions which will result in two completely different net lengths for \textit{b}. On the other hand, if net \textit{b} is routed before \textit{a}, the two optimal solutions for \textit{b} will not affect the optimal solution for \textit{a}.

![Figure 3.13: Possible solutions when net a is routed before net b.](image1)

![Figure 3.14: Possible solutions when net b is routed before net a.](image2)

Even if the routing area have multiple layers so that wire crossovers are allowed, net ordering is important, as it is usually desired to find a solution with as few wire crossovers as possible.

There are several methods to decide in which order the nets should be routed. In some cases, it may be enough to order the nets in the ascending order of the number of terminals within their bounding boxes or order the nets in the ascending order of the distance between source and target terminals. Other designs may require the order to be based on the nets’ timing criticality.

![Figure 3.15: Routing ordering decided by the number of terminals within the bounding boxes: a(0) --> b(1) --> d(2) --> c(6)](image3)
3.1.7 Channel routing

Channel routing [9] has been one of the main types of routers used in 2-layer VLSI designs. A channel consists of two rows of terminals. Each net may contain some terminals on the upper and lower row. Between the channel boundaries there are some horizontal tracks to which the horizontal wire segments can be assigned.

Traditionally, two layers are available for routing and the Manhattan model is adopted, i.e. all horizontal wire segments are routed in one layer and all vertical segments in the other layer. Wire segments on different layers that cross each other do not connect unless an explicit connection called a via is constructed to connect between layers.

![Channel routing example](image)

**Figure 3.1.16:** a) Channel routing example. b) Terminology for channel routing.

Channel routing is about to find a feasible solution to assign the horizontal net segments into tracks so that the number of tracks used is minimized. Vertical net segments and vias will then be placed to connect horizontal segments of the same net in different tracks, and to connect the terminals of the net to horizontal segments of the net.

**Channel routing constraints**

A channel routing problem can be characterized by two constraints graphs; a horizontal constraint graph (HCG) and a vertical constraint graph (VCG). In order for the channel router to success, neither of these constraints must be violated.

The HCG indicates possible ways of assigning nets into one track and shows whether the horizontal span of two nets overlaps each other. HCG \( G = (V, E) \) is an undirected graph where \( V = \{v_i : v_i \text{ represents a net } n_i \} \) and \( E = \{(v_i, v_j) : \text{ a horizontal constraint exists between } n_i \text{ and } n_j \} \). In other words, each vertex in the graph \( G \) represents a horizontal net segment and an edge between \( i \) and \( j \) indicates that net \( i \) overlaps net \( j \).

![Channel routing problem and its HCG](image)

**Figure 3.1.17:** A channel routing problem and its HCG.
The VCG shows whether there exists a column such that the terminal on top of the column belongs to one net and the terminal on bottom of the column belongs to another net. VCG \( G = (V, E) \) is a directed graph where \( V = \{v_i : v_i \text{ represents a net } n_i\} \) and \( E = \{(v_i, v_j) : \text{ a vertical constraint exists between } n_i \text{ and } n_j\} \). In other words, each vertex in the graph \( G \) represents a vertical net segment and an edge from \( i \) to \( j \) indicates that net \( i \) must be assigned to a track above net \( j \) so that vertical connections do not overlap.

![Figure 3.1.18: A channel routing problem and its VCG.](image)

The channel routing problem with both horizontal and vertical constraints is an NP-complete problem according to [11].

**Left Edge Algorithm**

Hashimoto & Stevens proposed the first channel routing algorithm, known as the Left Edge Algorithm (LEA) [9][10]. In LEA, the two-layer horizontal-vertical model is used and it will generate an optimal solution for problems without vertical constraints. The time complexity for LEA is \( O(n \log n) \), where \( n \) is the number of nets, but can be improved to \( O(n \log d) \) with proper implementation, where \( d \) is the channel density (the number of tracks required). However, \( d \) is equal to \( n \) in the worst case.

LEA treats each net as an interval. The intervals are then sorted in increasing order of their left end \( x \)-coordinate, hence the name. For each net, according to the interval order, tracks are scanned from top to bottom, and the net is assigned to the first track that can accommodate the net without causing overlapping nets segments. If the net cannot be assigned to any existing track, a new track is created and the net is then assigned to it.

![Figure 3.1.19: a) Sort nets by left end points. b) Place nets greedily.](image)

Based on LEA, many useful channel routing algorithms has been developed [11], some of them incorporating vertical constraints. A simple example of a Left-Edge algorithm with vertical constraints is illustrated in Figure 3.1.20 below, where a vertical constraint graph is used to determine which nets that can be assigned to a specific track.
Dogleg Channel Router

A drawback with the basic Left-Edge algorithm is that it cannot handle cases with constraint cycles. The solution is to incorporate doglegs that will resolve these kinds of problems.

![Diagram](https://via.placeholder.com/150)

**Figure 3.1.21:** Doglegs are used to resolve constraint cycles.

Another reason to use doglegs is that it can help minimizing the number of required tracks, one of the channel router’s main objectives. The basic Left-Edge algorithm will put the entire net on a single track, but by using doglegs, it is possible to split a net and put it on several different tracks. However, using doglegs might incur penalty for additional vias.

![Diagram](https://via.placeholder.com/150)

**Figure 3.1.22:** a) No doglegs. b) With doglegs.

Constructing a simple dogleg channel router is easy. Each multi-terminal net is broken into a set of two-terminal nets. Horizontal constraints apply as usual, with the exception that consecutive two-terminal subnets of the same net can be placed adjacent to each other on the same track. Vertical constraints apply separately on each two-terminal subnet. See **Figure 3.1.23**.
3.1.8 Switchbox routing

The switchbox is a four-sided rectangular channel area, with nets entering and leaving at any side. A switchbox router is aimed at connecting net terminals within such a switchbox.

Several algorithms exist that handle switchbox routing [12] and there are channel routers with special routing schemes to avoid switchbox problems [13]. However, none of these algorithms will be described here as they are out of scope of this thesis. It is not required to understand these algorithms in order to follow the implementation description in section 3.2.

3.1.9 Nets in SyncSim

A net in SyncSim consists of two parts; a set of connections in the simulator and a bus design component for the graphical appearance (optional). Each simulator connection is a property to the simulator and connects one output port to one input port. In SyncSim jargon, a port is equivalent to a terminal and a bus is equivalent to a wire.

The simulator connections are defined in the design XML file as in the following example:

```xml
<simulator class="syncsim.simulators.softsim.SoftSim">
  ...
  <!-- Counter to val_Register, Compare -->
  <property Counter_out:in="Counter:val" />
  <property val_Register:in="Counter_out:out" />
  <property Compare:val="Counter_out:out" />
  ...
</simulator>
```

The XML code snippet above will connect the net shown in Figure 3.1.24 below. The component named ‘Counter’ has one output port ‘val’ that is connected to the input port ‘in’ of the component ‘Counter_out’. The output port ‘out’ of ‘Counter_out’ is in turn connected to the input port ‘in’ of component ‘val_Register’. ‘Counter_out’ is in this example a bus component.
The bus design component is constructed similar to any other components used in the design. It consists of a Part and one or several Components. A Part contains the functionality description of the simulated component and can extend either SoftPart or SoftSyncPart depending on whether the part should be combinatorial or synchronous. A Component is the graphical representation of the simulated component and must extend SSComponent, the base class for all graphical components in SyncSim. For the example design shown in this section, the bus logic is implemented in the java class file 'Example_BusPart' and its graphical representation is drawn in the java class file 'Example_BusComponent'. The XML code snippet below visually creates the net shown in Figure 3.1.24.

```xml
<component name="Counter_out" class="example.Example_BusPart">
  <gfx class="example.Example_BusComponent" width="30" height="1" x="20" y="65" />
  <gfx class="example.Example_BusComponent" width="1" height="45" x="20" y="20">
    <property orientation="vertical" />
  </gfx>
  <gfx class="example.Example_BusComponent" width="240" height="1" x="20" y="20" />
  <gfx class="example.Example_BusComponent" width="1" height="45" x="260" y="20">
    <property orientation="vertical" />
  </gfx>
  <gfx class="example.Example_BusComponent" width="70" height="1" x="230" y="65" />
  <property bitWidth="4" />
</component>
```

Each 'gfx' tag represents one bus segment in the net. The width, height, x and y attributes sets the size and location of each bus segment. Default orientation is horizontal, but it can be changed by setting an orientation property. A bitWidth property is used to set the bit width of the net signal.
3.2 Implementation

In this section, the implementation of the routing extension module in SyncSim will be described and explained. No actual code will be presented here. The interested reader is referred to the CVS repository [3] containing both the source for SyncSim as well as the router module. Javadoc documentation can be generated from the source code and may be consulted during perusal of this section. The router module was implemented using SyncSim 2.1 as a code base. Unless stated otherwise, all referred class names are new class files that are part of the router module.

3.2.1 Prerequisites

In order to implement a general routing module that can be used with any design, SyncSim must be able to provide the router with a floorplan, describing the location and placement of all components in the design, and a net list with information on all port connections that has to be made. The router must in turn give back a routing solution to SyncSim that can be inserted into the current design.

3.2.2 Providing the floorplan

A floorplan with locations and placements of all components in the design area must be given as an input to the router. This information is essential as it tells the router which areas that are occupied by components and which areas that are free for wire routing. As SyncSim is written in Java and uses Java’s Swing GUI framework to draw components, the floorplan is already available in the form of SyncSim’s workspace panel. This panel is a Swing JPanel object and holds information on each visible design component in the form of SyncSim SSComponent objects.

3.2.3 Providing the net list

The net list tells the router how the ports in the design should be connected. This information is available in the form of connection properties for the simulator. However, in the original SyncSim, ports are only defined by their name and stored as String objects that are mapped to their targets. There is no information about a port’s location or placement relative to other ports. Nor are there any other port properties available that are required by a router.

A new port representation: Port

In order to add essential attributes to a port, a new port representation object has been evolved. An abstract Port class is used to represent all kinds of ports and provides the following fields:

- Bit width
- Type
- Relative position
- Relative x-coordinate
- Relative y-coordinate
- Input
- Outputs
The bit width of the port is of no importance for the router, but as it is closely related to the port in other aspects it is added to make the port representation complete. The type field holds information on whether the port is an input port or an output port, or neither of these. Each port resides within some bounding box, e.g. a design component, and the relative position decides on which edge on the bounding box the port turns out to be. The port can be assigned to either the north, south, east or west side of the bounding box, or null if the position is unknown. The x- and y-coordinates sets the location of the port relative to the top-left corner of the bounding box. The input field is a reference to the port that feeds the current port with a signal, and the outputs field is a list of references to ports that the current port should feed with a signal.

SSPort

The SSPort extends the abstract Port class and represents an input or output port on a design component. Each net in a design is represented by a source SSPort in one end, and one or several target SSPorts in the other end(s). Besides the inherited fields from the Port class, SSPort also contains the following fields:

- SSComponent host
- Name
- Source
- Targets
- Route enable
- Visibility
- Routed

The SSComponent host is a reference to the graphical representation object of the design component to which the current port belongs. In the case of the SSPort, the SSComponent object will decide the bounding box for the port. The name field contains the name of the port. The source field is a direct reference to the source SSPort, i.e. the start port of a net. The targets field is a list of direct references to target SSPorts, i.e. end ports of a net. It is not always the case that a user wants all ports to be routed, so the route enable field will state whether the port should be routed or not. In order to hide a port, a visibility field is provided that can be set to either true or false. The routed field is a status field that indicates whether the port has been successfully routed or not.

Structure of a net list

The net list that will be provided to the router is an array of Net objects, where each Net represents a connection between a target SSPort and a source SSPort. There can be several Nets with the same source SSPort, but the target SSPort must be unique as each target may have only one source.

3.2.4 SSBus

In the end of the routing procedure, the routing module must present a routing solution to SyncSim that can be inserted into the current design. Previously, all nets in a SyncSim design has been build up by bus components defined separately within each design as ordinary design components. However, in order to create a general routing module that should be able to route nets in any design, there has to be some common way to handle bus components. It was decided to develop a general bus component, the SSBus. The routing solution from the
router will consist of a set of SSBuses, each of them representing one net segment, which will be added directly to SyncSim’s workspace panel.

Originally, bus components have been simulated in the same way as any other design component, but the SSBus is a bit different. It is only a graphical representation of a net segment and not part of the simulation. Still, it is possible to view the signal status of any SSBus net segment. Each SSBus has a reference to its source SSPort, from which the current signal status is fetched each time the component tooltip is shown.

An SSBus net segment can have several shapes. It is always based on either a horizontal or a vertical straight line, but the line endpoints can have any combination of an ordinary line endpoint, a knob or an arrow. When an SSBus is created by the router, it will set the appropriate line end type depending on the endpoint context. If the endpoint connects to an input port, it will have an arrow towards the port to illustrate the signal direction. If the endpoint connects to more than one other segment, it will have a knob on that end, indicating the net is branching off.

### 3.2.5 Interfacing with SyncSim

The routing module contains a `Router` class, which is the main class that ties the module together. It provides an interface to control and configure the routing system. The Router operates on a Java AWT `Container` object where it looks for components to route according to the router net list. The Router sees the Container object as the floorplan of the design, which in the case of SyncSim is the same as the workspace panel. The router net list is build while a design is loaded and then supplied to the router.

![SyncSim loaded with the router module enabled.](image)

**Figure 3.2.1:** SyncSim loaded with the router module enabled. Note the small icon button in the bottom left corner, which opens the router configuration window.

The route handler

The `RouteHandler` class acts like a ‘brain’ to the routing system and decides how and when routing should be executed. It will add listeners to the router’s container in order to respond to user events and reroute the design when necessary. The RouteHandler also adds itself on
top of the Router’s container to act as a display component for router status messages and errors, as illustrated in Figure 3.2.2 below.

![Figure 3.2.2: A misplaced design component causes the router to fail. A warning message shows up at the bottom of the router’s container.](image)

3.2.6 Router configurations

Several router configuration options can be accessed via the SyncSim GUI. By clicking on the small icon in the bottom left corner of the SyncSim workspace panel, the router configuration window opens. See Figure 3.2.3.

**Wire turn penalty**

The router will always try to find the shortest path from source to target. However, in many cases it is desirable to get a simple path with as few turns as possible rather than a short path. By setting a wire turn penalty, the router will treat each wire turn as an additional length. As a result, less wire turns will be taken and the path may be simpler.

**Wire crossover penalty**

Setting a wire crossover penalty may help minimizing the number of wire crossovers. The router will treat each wire crossover as an additional length, which may open up for other paths that do not cross wires.

![Figure 3.2.3: Router configuration window.](image)


**Wire crossover type**

The wire crossover type will decide how wire crossovers should be displayed.

- **Mixed:** Draws the wires in the order they are created, causing a mix of vertical wires on top of horizontal and vice versa.
- **Vertical on top:** Divides the wires in two layers where vertical wires are put in the layer on top of horizontal wires.
- **Horizontal on top:** Similar to ‘Vertical on top’ but will put horizontal wires in the layer on top of vertical wires.
- **Same level:** Draws wires as if they were all in the same layer, causing no gap between vertical and horizontal lines.

**Preferred minimum wire distance**

The router uses the preferred minimum wire distance value when it decides how many wires that can be put in one channel. A small wire distance makes it easier for the router to find a short path, but it may clutter some channels with many wires and make it hard to follow the wiring. A large wire distance distributes wires on more channels, which may be easier to follow, but it might also come to a point where the router is unable to find room for any more wires and the routing will fail.

**Default component halo**

Reserves some space outside a component where no wire can be routed in parallel to the component. Only wires perpendicular to the component may pass through the component halo.

**Router status messages**

Enables or disables router status messages. Note that router error or warning messages cannot be turned off.

**3.2.7 Preparing to route**

The routing procedure is executed in several steps. To make things a bit easier to handle, the router will not take any pre-routed nets into account. Whenever some part of a design needs to be routed, everything will be rerouted. Thus, the first step to prepare to route is to clean up after any previous route. Cleaning up a previous route involves removing all SSBus components from the routing area, i.e. SyncSim’s workspace panel, and to disconnect any routing connections in the net list.

The second step is to sort the net list in ascending order of the Manhattan distance between target and source ports. By sorting the net list, the total net length may be minimized, or at least be kept shorter than if the net list order were unsorted. See section 3.1.6 for more info about net ordering.

After cleaning up any previous routing and sorting the net list it is time to hand over to the actual routing algorithms. The implementation of the routing module uses the same strategy as described in section 3.1.2, namely to split the problem into global routing and detailed routing.
3.2.8 Global routing

The *GlobalRouter* takes care of the global routing step. Its task is to find the paths for all nets in the router net list. The GlobalRouter will do this by creating a router *Maze* and running a maze routing algorithm (see section 3.1.5) on each net in the net list.

Creating a maze

A *Maze* object is defined as a simplified floorplan that describes a design in a matrix of two kinds of building blocks; *ChannelBlocks* and *SSBlocks*. The partitioning of ChannelBlocks is decided by the edges of the design components, i.e. the SSComponents, on the main floorplan. Each SSComponent is then represented by an SSBlock, added on top of the ChannelBlocks. Wires are routed from one SSBlock to another SSBlock and wires can only go through the ChannelBlocks.

![Diagram of Maze and SSComponents](image)

**Figure 3.2.4:** The SSComponents on the main floorplan for the example design.

![Diagram of Maze and ChannelBlocks](image)

**Figure 3.2.5:** The corresponding Maze floorplan divided into ChannelBlocks (illustrated by empty rectangles) and SSBlocks (illustrated by rectangles labeled ‘SSB’).
In theory, the floorplan can be infinitely large, but in order to handle it in practice, the area for the ChannelBlocks has to be restricted. As default, the Maze floorplan bounding box is calculated from the area required by the SSComponents, extended Maze.SOFT_BOUNDS_HALO pixels in each direction. However, if the default bounding box is not suitable, it is possible to specify a bounding box for the Maze floorplan that overrides the default.

**ChannelPort**

The *ChannelPort* extends the abstract Port class and represents an input or output port on a ChannelBlock. Besides the inherited fields from the Port class, ChannelPort also contains the following fields:

- ChannelBlock host
- Searching

The ChannelBlock host is a reference to the ChannelBlock to which the current port belongs. In the case of the ChannelPort, the ChannelBlock object will decide the bounding box for the port. The Searching field is a semaphore flag that is used when searching for port coordinates during detailed routing (see page 33).

Besides setting up a matrix of ChannelBlocks and SSBlocks, creation of the Maze floorplan also involves setting up an initial set of ChannelPorts that connects ChannelBlocks to SSBlocks. The result is illustrated in Figure 3.2.6 below:

![Figure 3.2.6: ChannelPorts at ChannelBlock edges connects to SSPorts at SSBlock edges.](image)

The arrow within each circle illustrates the signal direction. Note that although the ChannelPorts are connected to the SSPorts, their exact location has not yet been determined.

Each ChannelPort is assigned to either north, east, south or west side of its ChannelBlock, depending on which side that adjoins to the connected SSBlock. Thus, one of the x- or y-coordinate for the ChannelPort is defined at this stage. However, the other x- or y-coordinate will not be determined until the detailed router takes over.
Breadth-first search

For each routable net in the router net list, the GlobalRouter performs a breadth-first search to find a suitable path between source and target port through the ChannelBlocks of the Maze floorplan. During this phase, current router configuration values is used to add penalties for wire turns and crossovers, and the channel capacity for each ChannelBlock is limited by component halos and minimum wire distances.

It is generally easier to use the target as a starting point rather than the source, because the target has only one source while the source can have several targets. Thus, the search algorithm starts from the net target SSPort and then proceeds to its input port, i.e. the connected ChannelPort. The ChannelBlock that contains the ChannelPort adjoins to the starting SSPort and its distance is set to 0.

From the ChannelBlock with distance 0, the search algorithm first scans the channel for any ChannelPort that is connected to the requested source SSPort. If such a port exists, a result object containing the distance to that ChannelPort will be returned. The distance through a ChannelBlock is the Manhattan distance from the entering edge of the block to the exiting edge of the block, plus any penalties for wire turns and crossovers.

If the source is not found within the current ChannelBlock, the breadth-first search expands its search tree in all possible directions, setting the distance for each visited ChannelBlock and adds them to a processing queue. A ChannelBlock will only be updated and added to the queue if the distance is not set or is less than its previous value. Each visited block will always have a reference back to its latest visitor, forming a minimum trace back path. The algorithm then repeats scanning for a connection to the source SSPort and expanding the search tree until all ChannelBlocks in the queue has been processed.

At the end of the breadth-first search from the target to the source, several routes may have been found, but only the shortest one is taken into account.

Figure 3.2.7: The breadth-first search starts out from the ChannelBlock closest to the target SSPort.
Figure 3.2.8: All ChannelBlocks in the breadth-first search queue has been processed. Each ChannelBlock has been assigned a distance value from the target SSPort, including penalties for wire turns and crossovers.

Wire crossovers

One important, but also rather complex, step during the breadth-first search is to determine whether to add a wire crossover penalty or not. In some cases, the decision is obvious, but in many other cases, it is only a matter of guessing whether a crossover is likely to happen. The main problem is that at the time the crossover penalty must be set, no other wires are actually completely routed. The router knows which nets that passes through a channel, but has not yet determined their exact locations.

Suppose we need to calculate the wire crossover penalty for a path between \( p_1 \) and \( p_2 \) through a ChannelBlock, the crossover detection algorithm used in the router implementation works as follows:

- Calculate the rectangle formed by \( p_1 \) and \( p_2 \).
- Scan the ChannelBlock for existing output ChannelPorts and calculate the rectangle formed by each output and their corresponding source SSPort.
- For each output-source rectangle that intersects with the \( p_1-p_2 \) rectangle, a wire crossover is likely to happen.

Figure 3.2.9: a) No overlapping rectangles – wire crossover is unlikely to happen.  
b) Overlapping rectangles – wire crossover is likely to happen.
Trace back
After each successful breadth-first search, the GlobalRouter enters its trace back routine. Starting from the source ChannelPort that was found during the breadth-first search, the GlobalRouter traces back to the target port and creates ChannelPorts along the way that connects the source with the target.

Figure 3.2.10: Tracing back to the target and creating ChannelPorts along the way.

3.2.9 Detailed routing
The DetailedRouter operates on the Maze floorplan returned from the GlobalRouter. A combination of a simple switch box router for each ChannelBlock and a modified LEA (Left Edge Algorithm) for all vertical aligned ChannelBlocks results in Tracks containing vertical net segments. The tracks are then positioned in relation to each other and assigned an exact location. After locations for vertical tracks have been determined, the DetailedRouter starts over and repeats its procedure for horizontal aligned ChannelBlocks.

BusPort
The BusPort extends the ChannelPort class (see page 30) and represents a port that will decide the exact location of an SSBus (see section 3.2.4). It is created by the detailed router during switchbox routing and the channel scan procedure. Besides the inherited fields from the ChannelPort class, BusPort also contains the following field:

- Wired

The Wired field indicates whether the BusPort has been successfully wired or not.

Finding port locations
Detailed routing is all about finding exact locations for ports. Thus, two frequently used methods during the detailed routing are `findX(int)` and `findY(int)`, defined in the abstract Port class. Whenever the router needs to find the location for a port, it will call either of these
methods, which will in turn ask connected, and aligned, ports for their locations and the port will update itself accordingly.

In many cases, the location of a port cannot be found because all connected ports have an unknown location and the location has to be decided by the router later on. However, if a port connects to an SSPort, the location can be set, as SSPorts always have a fixed location.

**Switch box routing**

Each ChannelBlock in the Maze floorplan can be seen as a switch box with nets entering and leaving in four directions or less. The first step for the DetailedRouter is to apply a switch box routing algorithm on each such ChannelBlock switch box. The main task is to find a rectilinear Steiner tree for each net and to create and locate BusPorts for each Steiner node.

A net path through a ChannelBlock is rather well defined and restricted. Each net enters the block at one edge and leaves the block at any of the other edges. A net can have only one input ChannelPort or output ChannelPort on each edge. If the switch box rotation is disregarded, seven different tree shapes are enough to cover all possible cases:

![Diagram of seven switch box routing cases](image)

**Figure 3.2.11:** Assume the switch box has been rotated to make the net always enter the ChannelBlock at the top edge.

- **a)** The net exits at the bottom edge. The location of the input and output ChannelPort do not interfere with each other. No BusPort is needed.
- **b)** The net exits at one or more perpendicular edge and possibly the bottom edge as well. No ChannelPort location will interfere with each other. One BusPort is needed.
- **c)** The net exits at the bottom edge and possibly any of the perpendicular edges. Only the input and the bottom output ChannelPort location do interfere with each other. Two BusPorts is needed.
- **d)** Same as c) but mirrored vertically.
- **e)** The net exits at both perpendicular edges, in any order, and possibly at the bottom edge as well. Only the perpendicular edge ChannelPort locations do interfere with each other. Two BusPorts is needed.
- **f)** The net exits at both perpendicular edges, in any order, and at the bottom edge as well. Both the perpendicular edge ChannelPort locations and the top and bottom ChannelPort locations do interfere with each other. Three BusPorts is needed.
- **g)** Same as f) but mirrored vertically.

The switch box router is implemented using some if statements that will match each net in the provided ChannelBlock to one of the Steiner tree shapes illustrated in **Figure 3.2.11** above. Steiner tree nodes, in the form of BusPorts, will be created accordingly.
During the switch box routing, the switch box router may be able to determine the location of some of the ChannelPorts and BusPorts if they are aligned to a fixed SSPort. In Figure 3.2.12 above, the y-location of the two ports closest to the source and target ports has been determined. All other port locations are still unresolved.

**Channel scan procedure**

The channel scan procedure scans all vertically (or horizontally) aligned ChannelBlocks for net endpoints and creates a list of Net objects, which, in this context, represents a vertical or horizontal net segment between two BusPorts. New BusPorts are created when necessary, e.g. replacing a ChannelPort that connects to an SSPort. Redundant ChannelPorts are removed.
Net merge
The net merge step merges any number of adjoining Nets into a single Net containing several BusPorts. By doing this, it is easy to make sure all of the adjoining net segments will be located on the same track. However, this also counteracts support for doglegs (see page 21). Doglegs might be useful in order to handle constraint cycles and to minimize the number of required tracks, but as it also may require more vias, it may result in a somewhat messier wire routing. As the main goal of the SyncSim router implementation was to create simple wire routing that is easy to follow, the decision was not to incorporate doglegs support.

![Net merge diagram](image)

**Figure 3.2.14:** In this example, three adjoining Nets are merged into one single Net containing all four BusPorts.

Net sort
In the Left Edge Algorithm (LEA), described on page 20, net segments are sorted according to their left end coordinate before assigning the nets to tracks. This gives an optimal solution when it comes to minimal track usage. However, in SyncSim, the channel router must be able to deal with net segments that may have a fixed location, and SSBlock halos, occupying parts of a ChannelBlock. Minimizing the number of wire crossovers is also preferred over a minimum number of tracks.

![Net sort example diagram](image)

**Figure 3.2.15:** Net sort example.

Consider the net sort example in **Figure 3.2.15** that illustrates a part of a channel. A channel in this context is defined as the complete set of horizontally aligned ChannelBlocks on one row of the floorplan. The problem is to find a suitable layout for horizontal net segments that connect all terminals within each net. There are three edges with terminals in this example. The terminal at the right edge has a fixed vertical location. The grey boxes represent areas occupied by SSBlock halos. Nets may only pass through these areas in a perpendicular direction.
Sorting the Nets is performed in three sorting steps. Firstly, the segments are sorted as in LEA, i.e. according to the left end coordinates. Then, all Nets are sorted by their preferred channel location. The preferred location is calculated as the center of the possible location interval for a Net. A Net that may have any location in the channel has its preferred location in the middle, but a Net that passes through a ChannelBlock, where part of the channel has been occupied by an SSBlock halo, can have a preferred location at either side of the channel. A Net with a fixed location has the fixed location as its preferred location. The sorting by preferred location is a stable sort, meaning that the relative order of Nets with equal values is preserved.

**Figure 3.2.16** illustrates the relative Net ordering after sorting by left endpoint. The table to the left lists the Nets in ascending order of the left endpoint. The left edge value is a length unit only used to illustrate the edge position in this particular example.

![Nets sorted by left endpoint](image)

**Figure 3.2.16**: Nets sorted by left endpoint.

**Figure 3.2.17** illustrates the relative Net ordering after sorting by preferred channel location. The table to the left lists the Nets in ascending order of the preferred distance from the top edge. In this example, the total channel height is assumed 70 length units. The preferred vertical location is affected by SSBlock halos and fixed vertical terminals.

![Nets sorted by preferred channel location](image)

**Figure 3.2.17**: Nets sorted by preferred channel location.

Finally, the Nets are sorted by alignment. The alignment value for a Net depends on the direction of input and outputs of the Net. For example, if a Net connects to one terminal port at the top of the channel and one terminal port at the bottom of the channel, the alignment value will be zero. If a Net connects to two terminal ports at the top and one terminal port at the bottom, the alignment value will be \(-2 + 1 = -1\), and it will be placed above any Net with an alignment value greater than \(-1\). **Figure 3.2.18** illustrates the relative Net ordering after sorting by alignment. The table to the left lists the Nets in ascending order of the alignment value.
Assigning Nets to Tracks

After the Nets have been sorted according to the routines described above, each Net is assigned to a track within the channel. Nets are placed greedily onto the tracks. Tracks are scanned from top to bottom, and each Net is assigned to the first track that can accommodate the Net without causing overlapping Nets or violating any other restrictions. The router implementation uses a Track object that is responsible for maintaining a list of assigned Nets and avoiding overlapping segments and mismatching locations. As an example, a Track containing a Net with a fixed location will not accept to add any other Nets that will interfere with that location. Nets without fixed location can be placed on any Track as long as there is no Net overlap. New tracks are created when necessary. Figure 3.2.19 illustrates the relative Net ordering after Track assignment. The table to the left lists the Tracks in the order they are created. Although the relative location is set, some Tracks may have an unknown absolute location.

<table>
<thead>
<tr>
<th>Track #</th>
<th>Net #</th>
<th>Vertical location</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 &amp; 4</td>
<td>?</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>?</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>40</td>
</tr>
</tbody>
</table>

Locating Tracks

When all Nets are assigned to Tracks, the preferred relative Net ordering is settled but the exact location of the Nets may still be unknown. During the Track location step, all non-located Tracks are given a fixed location. Track locations are calculated so that the Tracks are spread out evenly over the free channel area. When suitable locations have been calculated, each of the non-located Tracks is assigned to one of these locations. The first non-located Track is assigned to the first free location, and so on. The relative order of non-located Tracks is always preserved during this track location step, but it may happen that the order of an already fixed Track will change, as
illustrated in Figure 3.2.20 (compared to Figure 3.2.19). The table to the left lists the Tracks in ascending order of the calculated distance from the top edge. In this example, the total channel height is assumed 70 length units.

<table>
<thead>
<tr>
<th>Track #</th>
<th>Net #</th>
<th>Vertical location</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 &amp; 4</td>
<td>17.5</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>28.75</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>51.25</td>
</tr>
</tbody>
</table>

Even if the result from the Track location step sometimes deviates from the previous preferred Net order, it is in many cases better to have the Tracks spread out evenly rather than trying to squeeze them together at one side of the channel in order to preserve the Net order.

The completed result of the example described above is illustrated in Figure 3.2.21.
3.2.10 Presenting the routing solution to SyncSim

Figure 3.2.22 below illustrates the result from the DetailedRouter after locating each track of net segments. The floorplan has been updated so that each target SSPort is connected to a source SSPort through a series of BusPorts. The net path has been determined by the GlobalRouter and the exact location of each BusPort has been determined by the DetailedRouter. Left to do is to transfer the routing solution to SyncSim and insert it into the current design.

Figure 3.2.22: Example design after locating the tracks. The DetailedRouter has been working both vertically and horizontally to form a complete set of nets built on BusPorts.

SSBus creation

The SSBus was introduced in section 3.2.4. It is a general bus component representing one net segment in the routing solution. For each target SSPort in the net list provided to the Router, the Router traces back to the source SSPort and creates an SSBus component for each pair of BusPorts along the way. The ‘Wired’ field in the BusPort object (see page 33) is used to avoid creating more than one SSBus component for a BusPort, e.g. when several targets traces back to the same source.

The list of SSBus components are sorted regarded to the current wire crossover type (see section 3.2.6) and then added to the SyncSim workspace panel. The result can be seen in Figure 3.2.23 below.

Figure 3.2.23: The routing solution as presented to SyncSim.
3.2.11 Design XML file modifications

The goal with the SyncSim router module was to automate the wire creation between design components so that the designer does not have to bother writing anything about wires/buses in the design XML file. However, the router will automatically add the routing solution to the design XML file so that it will be possible to load all buses even if SyncSim is started with the router disabled.

Simulator net list

Consider the XML code snippet shown in section 3.1.9 which defines a net connection in the original SyncSim:

```xml
<!-- simulator -->
<simulator class="syncsim.simulators.softsim.SoftSim">
...
<!-- Counter to val_Register, Compare -->
<property Counter_out:in="Counter:val" />
<property val_Register:in="Counter_out:out" />
<property Compare:val="Counter_out:out" />
...
</simulator>
```

After the implementation of the router module, buses are no longer simulated and do not have to be a part of the simulator net list. Design component ports are directly connected to each other. Thus, the new version of this XML code snippet will be:

```xml
<!-- simulator -->
<simulator class="syncsim.simulators.softsim.SoftSim">
...
<!-- Counter to val_Register, Compare -->
<property val_Register:in="Counter:val" />
<property Compare:val="Counter:val" />
...
</simulator>
```

Ports

The router module implementation has introduced Port objects to SyncSim. See section 3.2.3 for details. The new Port objects also have an impact on the design XML file as an optional port tag within the component tag. Port names and types are still defined in a component’s ‘Part’ class file, but the port tag in the XML file makes it possible for the designer to add additional port attributes such as a relative location and whether it should be routed or not.

The tag element name is ‘port’ and the available attributes are:

- Required:
  - name: The name of the port
  - type: The type of port, either ‘in’ or ‘out’

- Optional:
  - x: The relative x-coordinate in pixels, or one of ‘east’ or ‘west’
  - y: The relative y-coordinate in pixels, or one of ‘north’ or ‘south’
  - visible: ‘yes’ if the port should be visible, otherwise ‘no’. Default is ‘yes’
  - route: ‘yes’ if the port should be routed, otherwise ‘no’. Default is ‘yes’
Even if the designer leaves the port tags out of the initial XML design file, they will be automatically added or updated by the router after a reroute.

The following XML code snippet shows how the port tag is used in the ‘Counter’ component in the example design

```
<component name="Counter" class="example.Example.CounterPart">
<gfx class="example.Example.CounterComponent" x="200" y="130" width="80" height="80">
  <property bitWidth="4"/>
  <port name="val" type="in" x="west" y="11" visible="yes" route="yes" />
  <port name="dir" type="in" x="west" y="64" visible="yes" route="yes" />
  <port name="val" type="out" x="east" y="11" visible="yes" route="yes" />
</gfx>
</component>
```

**Router configuration**

The router configuration are design specific and is added to the design XML file as in the example below:

```
<router>
  <property wireCrossoverType="0"/>
  <property wireCrossoverPenalty="100"/>
  <property wireTurnPenalty="350"/>
  <property wireMinDistance="5"/>
  <property statusMessages="1"/>
  <property defaultComponentHalo="10"/>
</router>
```

The available router configuration properties are:

- **wireCrossoverType**
  
  0 = mixed (default), 1 = vertical on top, 2 = horizontal on top, or 3 = same level

- **wireCrossoverPenalty**
  
  Penalty for wire crossovers. Any integer between 0 (default) and 500

- **wireTurnPenalty**
  
  Penalty for wire turns. Any integer between 0 (default) and 500

- **wireMinDistance**
  
  Preferred minimum wire distance. Any integer between 3 (default) and 20

- **statusMessages**
  
  0 = disable router status messages, 1 = enable router status messages (default)

- **defaultComponentHalo**
  
  Reserved space for component halo. Any integer between 3 (default) and 20

**Nets**

The bus components that originally was design specific and defined in the XML file as any other simulated component is now obsolete and replaced by a net definition created by the router. The following three XML code snippets define the nets shown in Figure 3.2.24 below.
Net #1
<net source="Counter:val" targets="Compare:val">
   <gfx class="syncsim.routing.SSBus" x="315" y="139" width="35" height="5">
      <property type="horizontal_arrow" />
   </gfx>
</net>

Net #2
<net source="Counter:val" targets="val_Register:in">
   <gfx class="syncsim.routing.SSBus" x="70" y="139" width="30" height="5">
      <property type="horizontal_arrow" />
   </gfx>
   <gfx class="syncsim.routing.SSBus" x="68" y="110" width="5" height="31">
      <property type="vertical" />
   </gfx>
   <gfx class="syncsim.routing.SSBus" x="70" y="108" width="245" height="5">
      <property type="horizontal" />
   </gfx>
   <gfx class="syncsim.routing.SSBus" x="313" y="110" width="5" height="31">
      <property type="vertical" />
   </gfx>
</net>

Net #3
<net source="Counter:val" targets="val_Register:in,Compare:val">
   <gfx class="syncsim.routing.SSBus" x="280" y="139" width="38" height="5">
      <property type="horizontal_knob" />
   </gfx>
</net>

Figure 3.2.24: The lines colored black are represented by the router as three net definitions in the design XML file.

All port, router configuration and net XML code is actually generated by the XMLWizard. The XMLWizard is closely related to the editing module. Thus, it is described in more details in Chapter 4, section 4.2.2.
3.2.12 SyncSim modifications

One of the objectives of the routing module was to make it as self-contained as possible so it can be used in any applications that might benefit from automatic routing. However, some changes had to be done in the SyncSim core files in order to incorporate the routing module in SyncSim.

The router module was implemented using SyncSim 2.1 as a code base. Below is a list of all existing SyncSim class files that were affected by the implementation of the routing extension module. For each class, a short description tells which functionality and/or responsibility that have been added to these files.

**synccsim.simulators.softsim.SoftSim**
- Finds target and source ports and adds the net (target – source) to the simulator net list

**synccsim.Loader**
- Loads router configuration properties
- Loads port attributes
- Loads routed nets

**synccsim.PartCapsule**
- Maintains a list of ports for the encapsulated part
- Provides methods for adding and getting ports

**synccsim.Simulator**
- Maintains a net list
- Provides methods for adding and getting nets

**synccsim.SoftPartCapsule**
- Creates and adds an SSPort for each port
- Provides a method for getting the signal assigned to an SSPort

**synccsim.SSComponent**
- Assigns ports to the graphical component, i.e. provides it with a bounding box
- Provides a method for relocating all ports associated with the graphical component, i.e. distributes the ports along the bounding box in an appropriate way
- Maintains a component halo that reserves space around the component where no wires will be routed in parallel to the component
- Overrides the setSize method and makes sure all ports are properly redistributed over the bounding box proportionate to the changed size

**synccsim.SyncSimFrame**
- Creates the router
- Initiates the first reroute when a design is loaded
- Sets up a route listener that will listen for any router events and update the source code pane when needed (see section 4.2.2 for details)

No changes have been done to any design-specific class file. The only change that affects current designs is the structure of the design XML file. See section 3.2.11 for further details.
3.3 Evaluation

This section will discuss some strengths and weaknesses of the implemented routing module as well as the value of the result in respect to the purpose of this thesis.

3.3.1 Router shortcomings

Wire crossover detection
The wire crossover detection is not 100% accurate because at the time the router must decide whether one wire path crosses any other wire paths, no other wire is actually completely routed. Thus, the router will add the crossover penalty to any crossover that is likely to happen, although it may not be a crossover in the end.

Minimum wire distance
The preferred minimum wire distance will have no effect on the detailed routing step when specific wire coordinates are calculated. A design component may have some ports closely together, which will force the wires to be closer than preferred minimum wire distance.

Doglegs
Constraint cycles are usually handled by doglegs (see page 21), but the implemented router does not support doglegs, thus it does not handle constraint cycles. However, constraint cycles can be avoided in most cases by just moving some component ports in the design.

If one which to add support for doglegs, it can be done by skipping the ‘Net merge’ step (see page 36), and then implement dogleg support as described on page 21.

3.3.2 Simulator speed

At least in theory, the simulator speed should have been improved because of the router implementation. This is because wires in a design are no longer part of the design simulation. Originally, all wires were bus components that were called by the SyncSim simulator each clock cycle. Now, the router creates SSBus components that fetch the signal value from the source port when necessary. The SyncSim simulator does not care about these SSBus components. Thus, fewer calls by the simulator during a clock cycle should mean faster execution. However, no practical tests have been done to prove this statement.

3.3.3 The design XML file

The implementation of the router module has eliminated the need for bus components in the design XML file as nets are generated automatically by the router. Thus, it is a lot easier for a designer to create a new design XML file for SyncSim. However, an XML file including a full routing solution is a bit larger than an old style design XML file as the net definitions created by the router requires more space than the corresponding XML code for bus components. An XML file including a full router solution also contains additional port tags with attributes.

3.3.4 SyncSim 3.0

The code base for this thesis was SyncSim 2.0. The implementations that has been done, such as the router module, has resulted in SyncSim 3.0. To make it backwards compatible with SyncSim 2.0 design files, and to avoid recreating nets in design files which will only be used
for simulating, it was decided to add a start parameter to switch between two SyncSim modes. If SyncSim 3.0 is started without this parameter, it will start in ‘simulator’ mode, which will disable the router. Only the nets that are already in the design XML file will be loaded. If SyncSim 3.0 is started with the parameter ‘edit’, it will start in ‘edit’ mode, which will enable the router. All nets will be rerouted automatically by the router and updated in the design XML file.

Compatibility
There are several existing design XML files prepared for SyncSim 2.0 using old style bus components. SyncSim 3.0 is fully backwards compatible with these design files if started in ‘simulator’ mode. Table 3.3.1 below describes compatibility status for three kinds of design XML file types:

- **Old** A SyncSim 2.0 design XML file with bus components
- **Simple** A SyncSim 3.0 design XML file without a routing solution.
  May or may not include port attributes such as port locations.
- **Complete** A SyncSim 3.0 design XML file with a routing solution and port locations.

<table>
<thead>
<tr>
<th>Design file type</th>
<th>Simulator mode</th>
<th>Edit mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old (SyncSim 2.0)</td>
<td>Fully compatible</td>
<td>Loads OK</td>
</tr>
<tr>
<td></td>
<td>Bus components are simulated</td>
<td>Routing will fail</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Simulates OK</td>
</tr>
<tr>
<td>Simple (SyncSim 3.0)</td>
<td>Loads OK</td>
<td>Fully compatible</td>
</tr>
<tr>
<td></td>
<td>No wires are displayed</td>
<td>Places ports automatically</td>
</tr>
<tr>
<td></td>
<td>Simulates OK</td>
<td>Routes nets automatically</td>
</tr>
<tr>
<td>Complete (SyncSim 3.0)</td>
<td>Fully compatible</td>
<td>Fully compatible</td>
</tr>
<tr>
<td></td>
<td>Loads pre-routed nets</td>
<td>Loads port locations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reroutes nets automatically</td>
</tr>
</tbody>
</table>

Table 3.3.1: Design XML file compatibility with SyncSim 3.0.

3.3.5 Test case – a routed MIPS design

From the very beginning, the router module was designed to be self-contained and as far as possible not dependent on SyncSim. This made it easy to test the router in a simple test application during development. Simple designs were created by adding mock SSComponents to a JPanel the same way as they would be added to the SyncSim workspace panel. Later on, when integration with SyncSim began, the SyncSim example design was used to test and debug the router. However, the goal with the router module was to be able to use it on any design of arbitrary size and complexity so some larger designs was necessary to confirm that the router module actually fulfilled its goal.

There exists a MIPS design made for SyncSim 2.0. For the final test of the router module, the choice was to convert the old MIPS design XML file into a SyncSim 3.0 design and test it with the implemented router. If the router could route the wires for the MIPS design in a satisfactory way, it would be considered as a strong indication that the router actually is capable of routing almost any design made for SyncSim in a satisfactory way.
The two MIPS designs are shown below. Figure 3.3.1 shows the original MIPS design made for SyncSim 2.0 and Figure 3.3.2 shows the routed MIPS design in SyncSim 3.0.

Figure 3.3.1: The original MIPS design made for SyncSim 2.0.

Figure 3.3.2: The routed MIPS design. The router uses the following configuration:
Wire turn penalty = 350, Wire crossover penalty = 100, Wire crossover type = vertical on top,
Preferred minimum wire distance = 5, Default component halo = 5.
Besides removing all bus components, the only modification that had to be done in the MIPS design XML file in order to make it routable, was to add a graphical component for each `SplitterPart` that was missing one. Figure 3.3.2 illustrates these as small rectangular blocks in the bottom left part of the design.

**Test case evaluation**

One might argue that the original, and manually routed, MIPS design is somewhat cleaner and simpler when it comes to the wire routing. Some wires in the router solution seem to be a bit off the ‘optimal’ solution. However, the solution can still be accepted as satisfactory.

A major benefit with the router solution is that signal directions and wire connections are clearer. Arrows are automatically created at the end of a wire and knobs are used to indicate a connection at a wire intersection. Signal direction arrows are indeed available in the old MIPS design as well, but then it is up to the designer to add these arrows manually as bus component properties. Obviously, that is not optimal.
Chapter 4

Editing

4.1 Background

This section describes some background information for the second goal of the second part of this thesis; to implement tools for editing design components within SyncSim.

4.1.1 The design XML file

Each design that will be simulated in SyncSim is represented as an XML file that describes the graphical composition and layout of a design. Design components are represented with a Part class for the functionality description and an optional Component class for a graphical visualization. Each design component may have port tags that set the relative location of each component port. The XML file also contains a net list for the simulator that tells how the component ports should be connected.

Figure 4.1.1 below shows the SyncSim example design loaded into SyncSim 2.0. The SyncSim GUI makes it possible to switch between a design view and a source code view of the loaded design. The source code view shows the contents of the design XML file, and the design view shows whatever is painted by the graphical components in the design.

![Figure 4.1.1: SyncSim example design. a) Design view. b) Source code view.](image-url)
4.1.2 Modifying existing designs
SyncSim 2.x offers only one way to modify an existing design. That is by editing the XML code in the source code view and then pressing the save and reload buttons in the SyncSim GUI. This method is equivalent to loading and modifying the XML file in an external editor before reloading it in SyncSim.

4.1.3 Problem description
The problem when creating or modifying an existing design is that the designer needs to spend a lot of time on writing XML code. During design development, the designer will most likely need to adjust the design layout several times in order to get it right.

The router module implementation (see Chapter 3), greatly reduces development time as the designer do not have to bother about coding XML for wire routing, but the designer is still responsible for the overall component layout. To further reduce the burden on the designer, tools are needed that will help the designer to edit design components without touching the XML code.

4.2 Implementation
In this section, the implementation of the editing extension module in SyncSim will be described and explained. No actual code will be presented here. The interested reader is referred to the CVS repository [3] containing both the source for SyncSim as well as the editing module. Javadoc documentation can be generated from the source code and may be consulted during perusal of this section. The editing module was implemented using SyncSim 2.1 as a code base. All referred class names starting with Edit are new class files that are part of the editing module.

4.2.1 Overview
The intention with the editing module is to present an alternative editing interface to the designer other than the raw XML code. The design XML code will still exist in its original form, and be editable through the SyncSim source code view or an external editor, but also through a more user-friendly interface. The new editing interface will modify the SyncSim workspace directly and then automatically update the XML code.

4.2.2 The XMLWizard
XMLWizard is a new helper class that is added to SyncSim’s base package. The XMLWizard takes care of updating certain parts of an XML file while keeping other parts intact, including empty lines and comments.

When a design is loaded into SyncSim, an XMLWizard is created and fed with XML data from the design XML file, element by element. Static elements that cannot or should not be changed, such as the XML header tag, XML comments and empty lines, are fed to the XMLWizard as text strings. Dynamic elements, on the other hand, that can be modified through the editing module are fed to the XMLWizard as object references. For example, a component port can be moved within the component bounds, i.e. its relative location can be modified. Thus, the XMLWizard has a reference to the corresponding SSPort object (see page 25) in the design. In the same way, design components are fed to the XMLWizard as references to corresponding design component objects.
Table 4.2.1 below lists all dynamic elements that are mapped to design objects.

<table>
<thead>
<tr>
<th>XML element tag name</th>
<th>Referenced design object</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>Design Capsule</td>
<td>Subclass of synsim.PartCapsule</td>
</tr>
<tr>
<td>gfx</td>
<td>Design Component</td>
<td>Subclass of synsim.SSComponent</td>
</tr>
<tr>
<td>port</td>
<td>Component port</td>
<td>syncsim.routing.SSPort</td>
</tr>
</tbody>
</table>

Table 4.2.1: Dynamic XML elements and their relations to design objects.

Element types
Each element fed to the XMLWizard is assigned to a type. There are currently three types of elements available:

- NORMAL
- ROUTER
- NET

The NORMAL type is the default type and is not treated in any special way. However, elements of type ROUTER or NET are handled a bit differently. The XML code for router settings and net definitions are supposed to be completely regenerated and replaced by the router module after each reroute. By marking existing router and net XML code with ROUTER and NET type respectively, the XMLWizard knows where to insert the updated router and net XML code. If there is no existing router or net XML code, it will be appended to the end of the XML file.

Keeping XML structure intact
The XMLWizard is designed to update the XML file while keeping the original XML structure and order intact as far as possible. The use of element types for marking certain parts of the XML code as described above makes it possible for the designer to write an XML file as desired, e.g. placing router configuration properties at the top, and the XMLWizard will not change that order when it updates the XML code.

XML output
When a design has been modified in any way, a call to the XMLWizard’s getXML method will generate an updated XML representation of the current design. The XMLWizard does this by iterating through its internal list of elements, which represents the original XML structure, generating and concatenating the code for each element. Static elements are output as is, while current values are fetched from the referenced objects for dynamic elements. Router XML code is fetched from the router module and existing elements marked ROUTER or NET is replaced, if available. Otherwise, the router XML code is appended at the end.
4.2.3 Edit handler

The editing module is only enabled when SyncSim is started in edit mode, i.e. started with the ‘edit’ parameter. The EditHandler class is written using the Singleton pattern [15] and is responsible for handling all component editing. When first created and initiated by SyncSim, it will quietly setup mouse listeners, prepare an EditGrid, and setup an EditToolbar. The EditHandler then remains silent and invisible until the user interacts with some design component. See Figure 4.2.1 below and section 4.2.4.

![Edit handler](image)

Figure 4.2.1: SyncSim 3.0 example design, loaded in edit mode.

a) Editing module is enabled but invisible and waiting for activation.

b) Clicking on a design component activates the edit handler which marks the selected component, shows the edit grid and shows the edit toolbar.

Edit grid

The EditGrid is a subclass of JComponent that can be added to any Container. It displays a grid of dots over the entire container area to be guidance for the designer when editing a design. The distance between the dots is the grid size and can be set by the setGridSnap method. The EditGrid provides several methods such as snapToGrid(Point) and snapToGrid(Rectangle) that snaps a given point or rectangle to the nearest grid dots.

The EditGrid can be setup to resize itself automatically along with its parent, i.e. its container. Alternatively, it may be updated by calling the update method.

Edit toolbar

The EditToolbar, shown in Figure 4.2.2, is a toolbar for the editing module and provides options for changing the edit grid size, enabling or disabling grid snap and enabling or disabling object snap.

![Edit toolbar](image)

Figure 4.2.2: Edit toolbar.
Zoom and pan support
The SyncSim workspace panel is, since the implementation of the zooming module (see Chapter 2) a ZCanvas container. In order to be useful, the EditGrid must therefore support both zooming and panning.

Zooming support is provided by implementing the ZoomListener interface and make the EditGrid listens for ZoomEvents. The grid can then rescale according to the current zoom level. As an example, if the grid is initiated with a grid size of 10 by 10 pixels, and then zoomed 200%, the grid dots will be painted as a 20 by 20 grid. To avoid a system slowdown when there is a high density of grid dots, e.g. when zooming out, no dots are painted if the distance is less than eight pixels. However, the snap-to-grid methods are still fully functional.

To support panning, the EditGrid keeps track of the upper left corner of the union bounds of all components on the container. The grid dots always start out from that point. Actually, the only component bounds which will be considered are the SSComponents’. Thus, pan support will not work unless there is at least one SSComponent on the container.

4.2.4 Selecting components to edit
The EditHandler maintains an internal list of design components selected for editing. Both single and multiple component editing are supported, but e.g. resizing is only possible on one component at a time.

Single selection is done by clicking on a design component with left mouse button. The component is unselected if clicked again, if any other component is selected or if the user clicks on the edit grid.

Multi selection is done by holding the shift key down while selecting components. The group of selected components is marked with a red bounding box outline. Single components within the multi selection can be unselected by holding the shift key down and clicking the component again. All components are unselected if clicking anywhere on the edit grid or on any component that is already selected.

Figure 4.2.3: a) Single selection. b) Multi selection.
4.2.5 Moving components

Each selected component is outlined with a red bounding box. Hovering over the bounding box with the mouse pointer changes the cursor to a two-axis move cursor. Whenever the move cursor is visible, it is possible to click and drag the mouse to move the selected component in any direction. If grid snap is enabled, the component edges snaps to the grid points. Otherwise, the component can be moved any distance, counted in pixels.

If multiple components are selected, it is possible to either move only one of the selected components or move all selected components at once. A single component is moved in the same way as described above. All components are moved at once if the user clicks and drags anywhere outside the selected components, but within the multi selection outline.

Any component movement triggers an EditEvent of type MOVED that is noticed by all registered EditListeners.

4.2.6 Resizing components

The bounding box outline of a selected component has eight handles used for resizing a component. The handles are shown as small red filled rectangles. Hovering over each of them with the mouse pointer changes the cursor to different kinds of resize cursors. The component can be resized in any direction depending on the used handle. As an example, the left edge handle is used to resize horizontally by moving the left edge of the component. The bottom right handle is used to resize both vertically and horizontally by moving the bottom and the right edge of the component. If grid snap is enabled, the component edges snaps to the grid points.

The resize function honors the minimum size defined by the component and uses this as a limitation of how small a component can be. The default is five pixels. Thus, a component can never be resized to a zero or a negative size.

Resizing is only possible for one component at a time. Multiple resize is not supported.

Any component resize triggers an EditEvent of type RESIZE that is noticed by all registered EditListeners.

4.2.7 Moving ports

If the selected component contains ports, they are illustrated as blue rectangles along the component edges. Hovering over each of them with the mouse pointer changes the cursor to a one-axis move cursor. A tooltip with the port name and type is also shown so that it is possible to identify the ports.

Ports can be moved along the component edges by click and drag on the port handle. A port can only be moved in one axis at a time, but by moving the port to a corner and change the mouse direction, it is possible to move the port onto the adjoining edge and change the axis of movement.

Any port movement triggers an EditEvent of type MOVE that is noticed by all registered EditListeners.
4.2.8 Object snap

The object snap is by default disabled but can be enabled via the edit toolbar. Object snap can be used to align ports and component edges.

Aligning ports

If object snap is enabled and a port is moved, the port snaps to the x location or y location of any other port within the current selection, if the port is moved within the object snap threshold. All reachable ports other than the ones on the same edge are possible snap locations, even if they are on another component within a multiple selection.

Consider the scenario shown in Figure 4.2.4 above. Object snap is enabled and port D is to be moved down along the left edge of the Compare component. There are two possible snap locations for port D, in line with A and F or in line with C. Port B is of scope of D. Port E is on the same edge as D and should never share the same location. Thus, object snap is not activated for E.

Port D can be moved smoothly along the edge until it enters a snap location zone. Then it snaps to the snap location and do not leave until the mouse cursor is dragged far enough away from the zone. Then the port moves smoothly again until it hits the next snap location zone and so on.

Aligning components

As with aligning ports (see previous section), the object snap is also useful for aligning components. However, the object snap is only active when moving component edges, i.e. when resizing. Object snap is not active when moving entire components or component groups.

The concept of object snap is the same for component edges as with ports. If object snap is enabled and a component edge is moved, the edge snaps to the x location or y location of any other component edge within the current selection, if the edge is moved within the object snap threshold. All reachable edges other than the ones on the same component are possible snap locations. In other words, object snap for component edges is only possible in a multi selection with at least two components.
4.2.9 Edit listeners

Any modifications to the design made by the edit handler trigger an EditEvent that is broadcasted to all EditListeners.

RouteHandler

The RouteHandler is part of the routing module (see Chapter 3, page 26) and responsible for deciding when to reroute. As the editing module was implemented, the RouteHandler was extended to implement the EditListener interface. This means that the router is notified whenever the design is modified by the editing module and it can take appropriate actions, i.e. perform a reroute.

Updating the XML source

When the router performs a reroute because of an edit event, the router triggers a route event that will make SyncSim use the XMLWizard (section 4.2.2) to update the SyncSim source code pane. Thus, the designer can modify the design in SyncSim using the graphical editing interface and the XML source code will be updated automatically, reflecting the modifications. SyncSim will indicate that the source has changed by enabling the save button in the SyncSim toolbar.

4.2.10 SyncSim modifications

One of the objectives of the editing module was to make it as self-contained as possible so it can be used in any applications that might benefit from editing tools. However, some changes had to be done in the SyncSim core files in order to incorporate the editing module in SyncSim.

The editing module was implemented using SyncSim 2.1 as a code base. Below is a list of existing SyncSim class files that were affected by the implementation of the editing extension module. For each class, a short description tells which functionality and/or responsibility that have been added to these files.

**syncsim.Loader**
- Feds the XMLWizard with element data while loading a design

**syncsim.SSComponent**
- Sets a default minimum size for the graphical component
- Creates an edit listener that listens for mouse events and calls EditHandler when mouse is clicked

**syncsim.SyncSimFrame**
- Creates the XMLWizard and makes it available to the Loader
- Creates and enables the EditHandler if started in edit mode
- Makes sure the EditHandler gets disposed when closing a design

No changes have been done to any design-specific class file.
4.3 Evaluation

This section will discuss some strengths and weaknesses of the implemented editing module as well as the value of the result in respect to the purpose of this thesis.

4.3.1 Result

The implementation of the editing module has resulted in a graphical user interface for modifying an existing SyncSim design without need for any XML coding. It is possible to move and resize design components as well as moving ports within a component. Grid snap and object snap is available to make component and port alignment easier.

4.3.2 Editing shortcomings

Creating designs is not supported

The editing tools are only able to modify existing design components. Creation of new components or ports is not supported. Thus, a designer must create an initial XML design file by hand before it can be loaded and edited in SyncSim.

Object snap

Object snap is only active when moving ports or when moving component edges, i.e. resizing. Moving entire components are not affected by object snap. However, grid snap works when moving components, so it may not matter that object snap do not.

Toolbar settings

Editing configuration such as grid size, grid snap and object snap is not saved between sessions. As long as the EditGrid and EditHandler remains in memory, all settings are preserved, but as soon as SyncSim is restarted, the configuration is reset to default values.

4.3.3 Design development time

Although it is not yet possible to create a new design from scratch using the editing tools, the designer can still save a lot of development time by creating a basic XML file containing all components and simulator connections, but without bothering about the design layout. The design file can then be loaded into SyncSim and, by using the editing tools, the designer can easily move around and resize components as desired to get a nice layout.
Chapter 5

Conclusions

The purpose of this thesis was to extend SyncSim with features needed both by teachers who use SyncSim for educational purposes and by the designers who create new designs for SyncSim. What was needed by teachers was zooming support, and designers needed layout-level design tools to reduce development time, as a lot of time was taken up by writing the layout XML code for the design. All of these goals were met, although there is of course always room for improvements.

**Zooming**

The zooming implementation gives the teacher, or any user, the ability to zoom into a design for a closer look or, if the design is larger than the visible area, zoom out to get an overview of the entire design. There are 17 predefined zoom levels in the range from 5% to 1000%, but the zooming system is flexible enough to support any zoom level.

Panning is a feature closely related to zooming. If the design is zoomed in, it must be possible to pan the design in order to move the focus. Thus, the original SyncSim workspace panel has been replaced with a panel supporting both zooming and panning. The workspace is no longer fixed in size but can be, at least in theory, infinitely large. The zooming and panning implementation do not limit the available size.

Unfortunately, the zooming implementation required a change in how design components were painted, so each `XXXComponent` Java file in existing designs had to be modified in order to work with the zooming feature.

**Routing**

The routing implementation significantly reduces development time for designers, as they do not have to bother about writing XML code for bus components. Wire routing is completely automated as long as the simulator net list is provided.

For future development, it would be advisable to test whether the router would benefit from adding doglegs support. There are also some issues with minimum wire distance and wire crossover detection that may need to be addressed. Some strange wire routings can still occur in certain circumstances, but lack of time has left them be without further investigations.

The routing implementation has introduced some compatibility issues with older SyncSim design files, as these contain obsolete bus components that prevents the routing to success. However, the old design files can rather easily be converted to the new format by removing all bus components and connections. For large designs this can take some time, so in addition
to the goals defined by this thesis, a converter tool has been developed that helps removing all bus components from old design files.

**Editing**

The editing implementation further reduces development time for designers, as they can setup the design layout using graphical editing tools instead of by coding XML.

The current implementation provides only some basic editing functions for moving and resizing existing design components. Future development may extend the editing module with tools for creating new components from scratch, setting up ports and creating/modifying the simulator net list. The ultimate goal would be if a design could be created from scratch within SyncSim, or a separate editing program, without writing a single line of XML code!
Bibliography

[1] SyncSim


[3] SyncSim CVS repository


[7] Zhou, Hai. Global routing: geometric spanning trees; Steiner trees; net ordering,


[10] Channel Router Applet: The Left Edge Algorithm


