Summer Fellowship Report

On

Porting and Packaging of Mixed-Signal Simulator (NGHDL) to MS Windows and Packaging of KiCad

Submitted by

Bladen Martin

Under the guidance of

Prof.Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

Bladen Martin

August 21, 2020

Acknowledgment

It is a pleasure to acknowledge the help and support that has gone to in making this successful project. I express sincere gratitude to Prof. Kannan Moudgalya and FOSSEE Team at IIT Bombay for providing me with this opportunity to work on this project and having faith in my abilities. I would also like to express our gratitude towards Prof. Madhav Desai for showing us a clear path to simulate the process. I would like to thank my mentors Rahul Paknikar, Saurabh Bansode & Gloria Nandihal, and fellow interns Yamini Vadissa, Sumanto Kar, Shubhangi Mahajan & Ashutosh Jha for helping me out with all the problems that I faced while porting NGHDL and making my fellowship experience an enjoyable one. Lastly, I would like to thank the managers Usha Viswanathan and Vineeta Ghavri for helping me with the communication and managing the meetings with the professors.

Contents

Chapter 1

Background

NGDHL is a mixed-signal simulator, a part of the eSim, a free/libre and open-source EDA tool for circuit design, simulation, analysis, and PCB design.

A mixed-signal circuit is one which involves both analog as well as digital components and is found in many practical use case scenarios, thus marking the need for a mixed-signal simulation tool.

NGHDL is built upon intercommunication between two open-source tools Ngspice and GHDL, which also give it its name. NGHDL has already been implemented for Linux Operating systems and a detailed report on the same can be found here.

The purpose of this fellowship is to port NGHDL to Microsoft Windows OS, owing to the large market share of Windows OS and its presence in commercial and educational institutes. The aim is to navigate through the differences of Linux and Windows operating systems, making sure the functionality and efficiency of NGHDL is maintained, meanwhile offering an easy to install and use experience for the end-user.

Chapter 2

Introduction

2.1 VHDL

VHDL is an acronym for Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (HDL), which is a hardware description language used to describe a logic circuit by function, data flow behaviour, or structure.

2.2 GHDL

GHDL is a shorthand for G Hardware Design Language (currently, G has no meaning). It is a VHDL compiler that can execute (nearly) any VHDL program. GHDL is not a synthesis tool: we cannot create a netlist with GHDL (yet). Unlike some other simulators, GHDL is a compiler: it directly translates a VHDL file to machine code, without using an intermediary language such as C or C++.

2.3 Ngspice

Ngspice is a mixed-level/mixed-signal circuit simulator based on three open-source software packages: Spice3f5, Cider1b1 and Xspice: Xspice is an extension to Spice3 that provides code modelling support and simulation of digital components through an embedded event-driven algorithm.

Since GHDL cannot generate netlist files for Ngspice simulations, the aim here is to generate code models using Xspice and simulate digital as well as mixed-signal simulation through embedded event-driven algorithm, in this case, socket programming between GHDL and Ngspice.

Chapter 3

Porting from Linux to Windows

Methods for porting a Linux application to Windows

3.1 Windows Subsystem for Linux (WSL)

Run the program as-is on the Windows Subsystem for Linux (WSL). X-server can be used for enabling GUI.

- Pros – In WSL, your program executes directly on the machine hardware, not in a virtual machine.
- Cons – This is not an actual Windows port. We are just running the already existing Linux application on Windows with WSL. Also, the end-user must install WSL manually in their system.This method will only work on Windows 10.

3.2 Cygwin

Compile and run the program using GCC or clang in the Cygwin environment, which provides an almost complete `POSIX` environment on Windows.

- Pros – Requires changing less amount of code from the Linux version. The Cygnal project re configures behaviour of certain functionality in order to conform with native conventions of the Windows platform, which will help with tasks such as file system handling.
- Compiling with Cygwin requires other libraries to be supplied with such as cygwin1.dll and any others that may be created.Since we are doing make and make install in Ngspice to load the code models, Cygwin must be supplied with NGHDL, which adds considerable size overhead.

3.3 MSYS2 + MinGW

Similar to Cygwin, MSYS2 also provides a `POSIX` translation layer and builds Windows applications from Linux sources with dependency on `msys-2.0.dll`.

3.4 LBW

Essentially Wine in reverse, LBW is a system call translator that allows you to run unmodified Linux `x86` binaries on Windows.(https://github.com/davidgiven/LBW)

- Pros – Like WINE (which allows native Windows applications to run on Linux), this allows Linux applications to run on Windows.
- Cons – However, it is very unstable, as mentioned by the developer himself. So, this option must be discarded.

3.5 UWIN

From AT&T Labs(https://github.com/att/uwin)

- Pros – UWIN allows `UNIX` applications to be built and run on several versions of Microsoft Windows, with few, if any, changes necessary.
- Cons – However last known development was in 2012 with very little documentation and support available. So, compatibility and stability will be an issue. Hence this option must be discarded.

3.6 Code Re-writing

Manually port your code from Linux and compile for Windows using MinGW GCC. This involves re-writing the Linux-specific code to use Windows-specific code. MSYS can supplement the deficiencies of the CMD shell with GNU utilities such as `bash`, `make`, `gawk` and `grep`. It is intended to supplement MinGW.

- Pros – The use of MinGW GCC compiler provides a translation layer from Linux to Windows system calls, and the resulting executable is native Windows application requiring no dependencies or libraries to be provided separately. Compatibility is maximum in this approach.
- Cons – Coding effort is required to change every Linux specific part to Windows.

Trials and Conclusion

- Cygwin couldn't build Ngspice with GUI.
- MSYS2 approach led to GHDL server application closing due to use of BSD sockets instead of Windows sockets.
- Code restructuring option with MinGW was chosen as a method to port NGHDL.

Chapter 4

Installing NGHDL

4.1 GHDL

We need to use either '`LLVM`' backend or '`GCC`' backend since we need to interface GHDL with C here, which needs `-Wl` flag, which requires either LLVM or GCC backend.
We choose to use the pre-built Windows executable package with LLVM backend offered on the GHDL website. This package is portable, requires no complicated installation procedure and can be used directly out of the box.
Furthermore, we add the location of ghdl.exe to the path environment variable of the user so that it can be called from any location.

4.2 Ngspice

Ngspice also has pre-built packages for Windows, which can be downloaded directly, but here we need to build from source again as we need to do two things -

1. Add GHDL code models, so that GHDL code can be executed at runtime.
2. Add a patch to handle orphan testbench processes.
   Orphan testbench processes are those simulations which have run until completion and plotted, but the testbench files are still executing in an infinite loop.

For adding GHDL codemodels, so that they get executed at runtime, we need to-

1. Add GHDL to GNUmakefile in the codemodels list.
2. Add GHDL code models to spinit file, to include them at runtime.(*Also create the appropriate directories for GHDL*)
3. While generating GHDL codemodel, link with libws2_32.a library as Windows sockets have been used on the client side.

For handling orphan testbench process, a log of testbench IP address and port numbers are stored in C:\Windows\Temp, under the filename NGHDL_COMMON_IP_PID.txt, where PID is the process ID of Ngspice.

After simulation and before result plotting, these IP addresses and port numbers are read, and a special kill message is sent to the testbench using socket programming, after which the testbench then calls its exit procedure. The patch can be found here.
Before building Ngspice, the above mentioned patches must be applied, and then it needs to be installed at the appropriate location.

4.3 MinGW Environment

- The MinGW-w64 environment is available as prebuilt package for direct download here.
  This package requires no installation procedure and is extracted in the appropriate location. After adding it to the location, the path /mingw64/bin/ is added to the path environment variable of the user, which enables us to access GCC from any location.

- Similarly the MSYS shell for supplementing MinGW is also available as a prebuilt package archive and can be downloaded here.
  This package is extracted in the /mingw64 folder and then the path /mingw64/msys/bin/ is added to the path environment variable of the user, which allows us to access bash, make & other utilities.

4.4 Others

Other dependencies are installed when we install eSim. They include Python and PyQt4. Also make sure to create a symlink if they are not created for Ngspice, GHDL or NGHDL.

Chapter 5

Build Environment

Since Ngspice is being built from source, `GNU` utilities like make, GCC and a Linux like shell which understands commands like `grep` and `uname` are required.
There are utilities like GNU make for Windows, but these cannot be used since most of the `makefiles` are written for Linux and will not work with make for Windows due to various reasons such as the absence of commands like `grep` and `uname`.

- For the compiler, GCC provided by MinGW-w64 project is being used. This compiler can create native Windows applications without any dependency on POSIX translation `DLLs`.

- The rest of the utilities GNU utilities such as `bash`,`make`, `gawk` and `grep`, which allow us to run Linux shell scripts are provided by the MSYS project which supplements the deficiencies of the CMD shell.

  This should not be confused with the MSYS (or MSYS2) development environment, which provide a POSIX translation layer for building native Linux applications on Windows, with a dependency on the msys-1.0.dll (or msys-2.0.dll).

- These utilities are added to the path environment variable of the user as mentioned above, so that they can be used together and from any location.

- Bash is used for executing shell scripts and building Ngspice, using the configure – make – make install process.

Chapter 6

Windows Sockets

Since socket programming is the backbone of NGHDL, it is vital to understand the differences between Windows and Linux (BSD) sockets.
Windows sockets are provided by the winsock2.h library.

- Windows sockets, unlike Linux sockets requires a startup function `WSAStartup(MAKEWORD(2,0), &WSAData);` which tells the OS we are going to use sockets. `MAKEWORD(2,2)` is for specifying the version of Winsock we want to use, i.e. winsock2.h
- The function `WSACleanup();` is to be used to let the OS know that we are done using sockets so it can free up computer resources.
- While building any Windows socket application, it must be linked with the libws2_32.a library.

Chapter 7

NGHDL Workflow

There are primarily two steps to simulating your VHDL code using NGHDL

1. Executing NGHDL generates codemodel and DUT files for Ngspice.
2. Executing Ngspice to exchange messages between GHDL and Ngspice (*This is where the socket programming is used.*)

7.1 Generating Codemodel (Ngspice) and DUT files

When Ngspice executes the netlist files, it parses the models used and searches for the same in its libraries. If we want to use our library, we can add those using codemodels which are supported by `Xspice`. In this case, the codemodel files are generated by NGHDL. NGHDL has a few steps as follows-

1. **Create model directory**
   The directory is created with the name of the model to be generated at - /ngspice-nghdl/src/xspice/icm/ghdl. This directory shall contain all the files generated for the particular mode.
2. **Adding model in modpath**
   The name of the model that is created is added at modpath.lst, can be found at - /ngspice-nghdl/src/xspice/icm/ghdl this contains the list of all the ghdl models. When make is getting executed for Ngspice, the modpath.lst is read and then respective models are generated to codemodel files.
3. **Generating model files**
   The `vhdl` file is parsed for its input and output ports and accordingly models are generated, this is done by /nghdl/src/modelgeneration.py. The files that are generated are -
   - connection_info.txt
   - cfunc.mod
   - modelName tb.vhdl

- start server.sh
- sock pkg create.sh

4. **Generate codemodel files**
   Finally `make` and `make install` are called for Ngspice codemodels at /ngspice-nghdl/release/src/xspice/icm, which goes through all the the codemodels (ghdl included), and then goes through their modpath.lst as well, and accordingly executes all the models mentioned, generates cfunc.c and links them under ghdl.cm in our case. Also, here the cfunc.mod file is converted to cfunc.c file format by passing the data provided by Ngspice, which is stored in `mif private`, the structure for this data can be found at-
   /ngspice-nghdl/src/include/ngspice/mifcmdat.h file

   `mif private` consists of all the data Ngspice sends, the live data as well as the limits, such as stop time for simulation, current time, current input loads, among others.

This is the first part of NGHDL, which generates the codemodel files as well as the DUT files that contain testbench file for the model mentioned.

The codemodels are places primarily at-

1. /ngspice-nghdl/src/xspice/icm/ghdl
2. /ngspice-nghdl/src/xspice/icm/ghdl/modpath.lst
3. /ngspice-nghdl/release/src/xspice/icm/ghdl

7.2 Ngspice - GHDL interaction

When the Ngspice simulation is called for digital or mixed model circuits, the following steps taking place -

1. cfunc.c file located at the release folder-
   /ngspice-nghdl/release/src/xspice/icm/ghdl/modelName is called, this is the client-side

2. Parameters are passed from Ngspice to this file, using mif private structure, and accordingly, parameters such as socketid, time-limit, load, among others are set.

3. The above step, analyses the .vhdl files and finally, elaborates and interfaces `ghdlserver.o` and `modelName_tb file`, and lastly executes the `modelName_tb.exe`

4. modelName tb file contains the calls to the ghdlserver file, namely-

   - Vhpi_Initialize()
     Initialises server, setup the socket, bind it, and start listening to it

- Vhpi_Send()
- Vhpi_Listen()
  This function also has the condition for when the kill message is received from the orphan testbench patch, the exit procedure is called.
- Vhpi_Get_Port_Value()
- Vhpi_Set_Port_Value()

Vhpi commands are used for interaction of VHDL code with C code.

5. This testbench file, initialize, listens and sends data to and from ghdlserver.c file using the Vhpi commands.

6. On the client-side (cfunc.c) meanwhile, time is checked for each event, and then accordingly data is loaded from mif private and sent to the server, and when time is over, END signal is sent by the client, which closes the connection.

7.3 Logs and Simulation Data

- Mintty is being used to display the client-server transaction logs which take place during the simulation. Thus is achieved by launching the simulation through Mintty at the appropriate location.

- The rest of the simulation data is displayed on the Ngspice application window.

Chapter 8

OS Independent codes

Different languages provide different methods to make the code independent of the operating system, thus allowing us to maintain one file for that code.
Some of these methods are used in NGHDL codes to make them OS independent.

- **C** : C provides different macros for detecting the OS and then executing specific code based on it. For example, for the `GCC` compiler.

```
#ifdef _WIN32
    //Windows specific code goes here
#elif _linux_
    //Linux specific code goes here
#endif
```

- **Python** : Python's `OS` library provides a method to detect the OS and execute related code.

```
if os.name == 'nt':
    #Windows specific code
else:
    #Linux specific code
```

- **Makefile** : As socket programming is being used on the client side, a command must be added in the makefile to link the `libws2_32.a` library when the codemodels are generated as well as when Ngspice is being built due to socket programming being used in closing of `orphan testbenches`. This linking should only happen in Windows, hence this method is being used.

```
ISMINGW = $(shell uname | grep -c "MINGW32")
ifeq ($(ISMINGW), 1)     #If yes
    LIBS = -lm -lws2_32 #then link with libws2_32.a
endif
```

Chapter 9

Termination of VHDL Testbench

9.1 Issue

Every time a mixed-signal circuit simulation is started in Ngspice, an associated Testbench and GHDL Server is started. To close this instance of GHDL server, the existing solution of using Signals and process ID in Linux was not portable to Windows. Hence a common and scalable solution for both operating systems was needed.

9.2 Methodology

Socket programming is used to terminate the associated Testbench and GHDL Server. Ngspice sends a special END message (**"CLOSE_FROM_NGSPICE"**) to the server after the end of the simulation.

In order to do that, Ngspice needs the `Port Number` and `IP address` of that GHDL Server. This parameters are stored in a file named NGHDL_COMMON_IP_PID.txt, where PID is the process ID of Ngspice. This file is created by cfunc.c and stored at C:/Windows/Temp.
Now as it is part of the same simulation, the PID remains same and hence so we can find out exactly the filename which store the ports and IP address of that server by looking for it in the C:/Windows/Temp directory.
The patch then iterates through all lines of Port numbers and IP addresses, as there can be multiple models being simulated and sends them a special message. An `if condition` in the server checks for this message when received and when true, calls the `GHDL server exit` procedure.

```
static void close_server()
{
        FILE *fptr;
        char ip_filename[48];

        #ifdef __linux__
            sprintf(ip_filename, "/tmp/NGHDL_COMMON_IP_%d.txt", getpid());
        #elif _WIN32
            WSADATA WSAData;
            SOCKADDR_IN addr;
            WSAStartup(MAKEWORD(2, 2), &WSAData);
            sprintf(ip_filename, "C:\\Windows\\Temp\\NGHDL_COMMON_IP_%d.txt", getpid());
        #endif

        fptr = fopen(ip_filename, "r");

        if(fptr)
        {
            char server_ip[20], *message = "CLOSE_FROM_NGSPICE";
            int port = -1, sock = -1, try_limit = 0, skip_flag = 0;
            struct sockaddr_in serv_addr;
            serv_addr.sin_family = AF_INET;

            /* scan server ip and port to send close message */
            while(fscanf(fptr, "%s %d\n", server_ip, &port) == 2)
            {
                /* Create socket descriptor */
                try_limit = 10, skip_flag = 0;
                while(try_limit > 0)
                {
                    if((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                    {
                        sleep(0.2);
                        try_limit--;
                        if(try_limit == 0)
                        {
                            perror("\nClient Termination - Socket Failed: ");
                            skip_flag = 1;
                        }
                    }
                    else
                        break;
                }

                if (skip_flag)
                     continue;
                serv_addr.sin_port = htons(port);
                serv_addr.sin_addr.s_addr = inet_addr(server_ip);

                /* connect with the server */
                try_limit = 10, skip_flag = 0;
                while(try_limit > 0)
                {
                     if(connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
                     {
                         sleep(0.2);
                         try_limit--;
                         if(try_limit == 0)
                         {
                             perror("\nClient Termination - Connection Failed: ");
                             skip_flag = 1;
                         }
                     }
                     else
                         break;
                }

                if (skip_flag)
                     continue;

                /* send close message to the server */
                #ifdef __linux__
                     send(sock, message, strlen(message), 0);
                     close(sock);
                #elif _WIN32
                     send(sock, message, strlen(message) + 1, 0);
                     closesocket(sock);
                #endif
            }

            fclose(fptr);
    }
    #ifdef _WIN32
        WSACleanup();
    #endif

    remove(ip_filename);
}
```

Listing 1: Client side patch

Chapter 10

Nullsoft Scriptable Install System (NSIS)

NSIS (Nullsoft Scriptable Install System) is a professional open source system to create Windows installers.

10.1 NSIS Plugins

A NSIS plugin is a way of extending installer's functionality by adding new functions.
NSIS plugins consist of one mandatory file (with the extension ".dll") and usually a header file (with the extension ".nsh").
NSIS searches for plugins in the **Plugins** folder under the NSIS install directory and lists all of their available functions. *!addplugindir* can be used to tell NSIS to search in other directories too.

Steps to install NSIS plugins -

- After downloading the plugin, unpack it. It may contains various files along with the `.dll` file and/or `.nsh` file.
- Then the "`.dll`" files must be put into the "**NSIS/Plugins**[**/platform**]" subfolder and the "`.nsh`" file into the "**NSIS/Include**" subfolder.
- Plugins with extension `.nsh` can be placed in the same dirctory as the NSIS script in which the plugin is to be used and can be included using the command *`!include "plugin.nsh"`*

10.2 Plugins used for eSim installer

- EnVar plug-in - Used to set environment variables required for the working of eSim and NGHDL.

- CPUFeatures plug-in - Can detect CPU features, number of cores and CPU vendor at run-time. Used to get number of CPU cores of target system to speed up installation of Ngspice.
- Nsis7z plug-in - Used to decompress a `.7z` file.
- ZipDLL plug-in - Used to decompress a `.zip` file.

Chapter 11

Packaging of NGHDL for MS Windows - Python Executable

Automation of NGHDL is done using Python scripts, which also provides the GUI for the application.

11.1 Issue

Python is an interpreted language, which requires a program known as interpreter to constantly translate and execute the code. This leads to problem scenario that is distribution of the specific Python version to end-user along with NGHDL. Not only does this cause a size overhead but also may lead to conflicting Python distributions on the end-user's system.

11.2 Methodology

To avoid this, a process know as **Freezing** is applied. Freezing code is creating a single executable file to distribute to end-users, that contains all of your application code as well as the Python interpreter and minimum required dependencies.
PyInstaller is used to freeze NGHDL. It is a Python package and can be installed using pip. To track dependencies we install all packages in a Python virtual environment. It can be installed and activated by

```
$ pip install virtualenv
$ Python -m virtualenv nghdl
$ source nghdl/Scripts/activate

$ pip install pyinstaller
```

After installing PyInstaller, we install some additional packages required by NGHDL

```
$ pip install --upgrade 'setuptools<45.0.0'
$ pip install --upgrade 'sip<5.0.0'
$ pip install <wheel_package_of_PyQt4>
```

PyQt4 Wheel can be downloaded from -
https://www.lfd.uci.edu/~gohlke/pythonlibs/#PyQt4
After installing all the required packages, we check if all dependencies are available using the command :

```
$ pip freeze
```

The following dependencies should be available -

- PyQt4
- sip
- altgraph
- future
- pefile
- pyinstaller
- pywin32-ctypes

Create Spec file using the command

```
$ pyi-makespec --onefile -n nghdl
<path_to_nghdl>/src/ngspice_ghdl.py
```

The spec file is the description of what PyInstaller is to do with the program. The option --onefile specifies PyInstaller to build a single file with everything inside.

Create a single file Windows executable (.exe) using the command -

```
$ pyinstaller -F --clean nghdl.spec
```

Verify whether all NGHDL src files (*.py) have been included in Analysis-00.toc file under the build folder generated by PyInstaller.

The file **nghdl.exe** is placed under the *dist* folder generated by PyInstaller. This is a standalone executable file.

NGHDL is then packaged with eSim using the Nullsoft Scriptable Install System. A modular approach is followed by maintaing a separte NSIS script for NGHDL named **installnghdl.nsi** and then including it the eSim installer NSIS script using the NSIS command -

```
!include "installnghdl.nsi"
```

All scripts and documentation of steps related to packaging of NGHDL are present at the Windows folder under installers branch of the NGHDL repository - https://github.com/bladen-martin/nghdl/tree/installers/Windows

Chapter 12

Packaging of KiCad for eSim

KiCad is an open source software suite for Electronic Design Automation (EDA). The programs handle Schematic Capture, and PCB Layout with Gerber output.

12.1 Issue

KiCad has a standalone installer file provided by the KiCad Developers Team for version 4.0.7 (kicad-4.0.7-i686.exe).
This installer is integrated into the eSim installer using NSIS method *ExecWait* which waits till the completion of this KiCad installer.
The size of original `kicad-4.0.7-i686.exe` is 705 MB. This causes a large size overhead for the eSim installer.

12.2 Methodology

The solution to this problem has been identified as removing the folder **'packages3d'** present at KiCad/share/kicad/modules. This folder is the maximum contributor to size of KiCad and has also been identified as non-essential for the working of eSim. Users may download this folder separately to enable the KiCad functionality of viewing components in 3D if required.

To repackage KiCad installer after removing the above mentioned folder, a reverse engineering approach is followed.

- First step is to install KiCad using the original installer provided by the KiCad Developers Team at any preferred location.
- After deleting the folder **'packages3d'** present at KiCad/share/kicad/modules, the folders **bin**, **lib**, **share** & **ssl** are to be compressed into file named **KiCad.7z** using the 7z compression tool. **KiCad.7z** is to be placed at the KiCad installation directory.

- Next step is to place the folder **NSIS** at the KiCad installation directory. This 'NSIS' folder can be obtained from the *executables* branch of KiCad-eSim git repository - `https://github.com/bladen-martin/KiCad-eSim/tree/executables`.

- After placing the folder at KiCad installation directory, the file **install.nsi** present inside the **NSIS** folder is compiled using the Nullsoft Scriptable Install System. This file is modified to create KiCad installer for eSim after making the above changes to reduce its size.

- The generated `kicad-4.0.7-i686.exe` in the **NSIS** folder is to be used during the development of eSim installer using NSIS as the default installer for KiCad.

12.3 Changes to *install.nsi*

Being an open source project, all KiCad source code can be obtained from KiCad's git repository.
After obtaining **install.nsi**, it is modified for eSim as follows -

- All commands adding files and folders to the installer i.e., "**File**" command, are commented out.

- The following code snippet which adds and extracts and deletes after extraction the above mentioned **KiCad.7z** is added to **install.nsi**.

```
File "..\KiCad.7z"
Nsis7z::ExtractWithDetails "$INSTDIR\KiCad.7z"
"Extracting KiCad %s..."
Delete $INSTDIR\KiCad.7z
```

Chapter 13

Conclusion

In the course of this fellowship the following tasks were achieved -

- NGHDL was successfully ported to MS Windows and packaged with eSim for distribution using open source utilities including but not limited to MinGW and MSYS.
- A new client-server architecture based patch was developed for closing of the GHDL server and all source code was made OS-independent wherever required.
- The size of KiCad installer executable was reduced, which intern led to the reduction in size of the eSim installer.

Reference

- NGHDL git repository : https://github.com/bladen-martin/nghdl
- Ngspice website : http://ngspice.sourceforge.net/download.html
- GHDL website : http://ghdl.free.fr/
- MinGW-w64 source : https://sourceforge.net/projects/mingw-w64/
- Nullsoft scriptable Install System Wiki : https://nsis.sourceforge.io/Main_Page
- KiCad website : https://kicad-pcb.org/
- Unofficial Windows Binaries for Python, by Christoph Gohlke, Laboratory for Fluorescence Dynamics, University of California, Irvine : https://www.lfd.uci.edu/~gohlke/pythonlibs/