eSim Semester Long Internship 2025-2026

On

Developing an Installer for eSim on macOS and Exploring Creating KiCad Plugins

Submitted by

Thet Htar Shwe Sin
Final Year Student
Computer Science and Engineering
Myanmar Institute of Information Technology

Under the guidance of

Prof. Prabhu Ramachandran
Department of Aerospace Engineering
IIT Bombay

May 4, 2026

Acknowledgment

I am grateful to the **FOSSEE team** for giving me this opportunity to contribute to the development of the eSim installer for macOS and for providing all the support and resources I needed throughout my internship. It has been a truly valuable experience.

I would like to thank **Prof. Prabhu Ramachandran** for his oversight and support of this internship, and **Prof. Kannan M. Moudgalya**, former head professor, whose efforts in arranging this opportunity made my internship at IIT Bombay possible.

I am deeply thankful to my instructor, **Mr. Sumanto Kar**, eSim Project Lead at FOSSEE, IIT Bombay, for his technical guidance, helpful feedback during weekly reviews, and patience in helping me understand the eSim codebase. His vision for making eSim accessible to a wider range of users, including macOS users, was the driving force behind this work.

I would also like to thank my internal mentors, **Mr. Varad Patil**, **Mr. Aditya Minocha**, and **Ms. Shanthi Priya**, for their support, technical inputs, and feedback throughout the development process. Their guidance helped me stay on track and work through the technical challenges involved in building the installer. My thanks also go to the broader eSim team for their help and guidance along the way.

I am grateful to **Mr. Piyush Kumar** from the Dean of International Relations (Dean IR) office for helping me and my fellow international students with our documentation and administrative needs, making our stay much smoother.

I would also like to thank my professor in Myanmar, **Mr. Joe Tun Sein**, for his role in helping us get this internship opportunity at IIT Bombay. His support and dedication to his students is something I truly appreciate.

I acknowledge my parent institute, Myanmar Institute of Information Technology, and my faculty there for their encouragement and support.

Finally, I thank my family and friends for their patience, belief, and motivation throughout this period. This accomplishment means a lot to me, and I could not have done it without them.

This internship lasted four months, one month working remotely from Myanmar and three months on-site at IIT Bombay. Working on the eSim macOS installer has been a meaningful and challenging experience that has grown my technical skills and broadened my perspective. I will carry what I have learned here with me going forward.

Contents

Chapter 1

Introduction

1.1 Background of EDA Tools

Electronic Design Automation (EDA) tools are essential for designing, simulating, and verifying electronic circuits. They support various stages of the design process, including schematic capture, simulation, and analysis, allowing designers to test circuit behavior before physical implementation.

As circuit complexity has increased over time, especially with modern integrated circuits, manual design methods have become impractical. EDA tools help manage this complexity by providing efficient and accurate design workflows. However, most advanced EDA tools are proprietary and expensive, which limits accessibility for students and researchers.

1.2 Importance of Open-Source EDA Tools

Open-source tools such as eSim provide an affordable and flexible alternative to proprietary EDA software, enabling users to explore circuit design, simulation, and analysis without licensing restrictions. They improve accessibility for students and researchers while also offering transparency and the ability to modify the software. Additionally, open-source platforms encourage collaboration and continuous development within the community.

1.3 Motivation

eSim is a free and open-source Electronic Design Automation (EDA) tool developed and maintained by the FOSSEE (Free and Open Source Software for Education) team at the Indian Institute of Technology Bombay. It provides a complete electronics design and simulation workflow, allowing users to draw schematics, generate SPICE netlists, run circuit simulations, and visualize waveforms — all within a single integrated environment. eSim is widely used in academic and research settings, particularly in Indian universities, as a free alternative to proprietary EDA tools such as Cadence and Multisim.

Despite its adoption across Linux and Windows platforms, eSim had no official macOS support prior to this project. The growing prevalence of macOS in academic institutions — and the rapid shift toward Apple Silicon hardware following Apple's introduction of the M-series chip architecture in 2020 — made this gap increasingly significant. Students and researchers using macOS machines had no practical path to running eSim without either setting up a virtual machine or manually compiling every dependency from source, a process involving ngspice, KiCad, GHDL, Verilator, and multiple Python packages. This was impractical for most users and entirely unsuitable for distribution at scale. This capstone project was undertaken to address that gap directly, with the goal of producing a complete, user-friendly macOS installer for eSim.

1.4 Objectives

The primary objective of this project is to develop a distributable macOS installer for eSim that requires no prior developer tooling on the part of the end user. Specifically, the project aims to:

1. Analyse the eSim codebase and its dependencies to understand the full requirements for a macOS port.
2. Establish a working development environment for eSim on Apple Silicon (arm64) macOS.
3. Build and configure ngspice 35 for macOS, including X11/XQuartz support for interactive plot windows.
4. Identify and apply all macOS-specific source code modifications required for correct behaviour on Darwin.
5. Bundle the complete eSim application — including its Python runtime, PyQt5 UI framework, ngspice binary, X11 shared libraries, and all supporting assets — into a self-contained macOS .app bundle using PyInstaller.
6. Produce a macOS .pkg installer that automates dependency checking, application installation, and initial configuration, requiring nothing more than a double-click from the user.

1.5 Scope

This project focuses on developing a complete and user-friendly macOS installer for eSim, aiming to simplify the installation process and improve accessibility for macOS users. It also explores enhancing integration between eSim and KiCad to support smoother design and simulation workflows.

In scope:

- macOS-specific modifications to eSim's Python source code
- Building ngspice 35 with X11 support and resolving its shared library dependencies for bundling
- PyInstaller configuration and all fixes required for a working frozen .app
- X11 dylib relinking and XQuartz integration
- KiCad symbol library registration for the bundled context
- Creation of a .pkg installer with preinstall and postinstall automation scripts
- Testing on Apple Silicon (arm64) macOS

Out scope:

- Intel macOS support (identified as future work)
- Apple Developer code signing and notarization
- GHDL bundling within the .app (identified as future work)
- Modifications to eSim's core simulation logic or feature set

Chapter 2

About eSim

2.1 Overview of eSim

eSim is an open-source Electronic Design Automation (EDA) tool developed under the FOSSEE initiative at IIT Bombay. It supports analog, digital, and mixed-signal simulations, providing a unified platform for circuit design and analysis. eSim integrates tools such as KiCad for schematic capture and Ngspice for simulation, allowing users to design, simulate, and visualize circuits within a single environment. It is widely used for educational and research purposes due to its accessibility, flexibility, and support for SPICE-based simulation workflows.

2.2 Architecture and Key Components

eSim is a Python application with a PyQt5 user interface. Its entry point is `src/frontEnd/Application.py`, which initialises the main window and coordinates the functional modules. All application logic is pure Python; external tools are invoked as subprocesses rather than linked libraries. The build system is Python setuptools and there is no CMake and no compiled C++ component within eSim itself.

The key source modules are:

- **frontEnd**: Main window, welcome screen, workspace management, and shared path utilities. This is the top-level module that orchestrates the rest of the application.
- **projManagement**: Project handling, KiCad subprocess launch (Kicad.py), and tool presence validation (Validation.py).
- **ngspiceSimulation**: Simulation process management, console output, and interactive plot window handling (NgspiceWidget.py).
- **nghdl/src/**: Contains the integration layer for GHDL-based VHDL simulation. The configuration for this module is stored in ∼/.nghdl/ on the user's machine.

- **kicadtoNgspice**: Converts KiCad netlists into ngspice-compatible format.
- **modelEditor**: SPICE device model editor.
- **subcircuit**: Subcircuit creation and management. :

All application logic is pure Python. External tools, ngspice, KiCad, GHDL, Verilator, are invoked as subprocesses rather than linked libraries. This modular architecture makes eSim flexible and extensible, but also introduces challenges when adapting it to different operating systems such as macOS.

2.3 Features

eSim provides a range of features that make it suitable for circuit design, simulation, and analysis. Some of the key features include:

- User-friendly GUI
- SPICE simulation support
- Multi-tool integration
- Open-source flexibility
- Schematic capture using KiCad
- Support for analog, digital, and mixed-signal circuits
- Custom component and library support
- Integration of eSim-specific symbols within KiCad
- Netlist generation for simulations
- Visualization of simulation results

2.4 External Dependencies

eSim is an integrated tool built using open source softwares such as:

1. **KiCad** – https://www.kicad.org/

   Provides the schematic editor. On macOS, its eeschema binary is embedded inside the application bundle at a non-standard path and is absent from the system PATH.

2. **Ngspice** – https://ngspice.sourceforge.io/

   The SPICE simulation engine, invoked as a subprocess. It depends on X11 libraries provided by XQuartz on macOS, and on code model plugin files (.cm files) that must be present at runtime. On macOS it must be compiled from source.

3. **GHDL** – http://ghdl.free.fr

   Handles VHDL simulation and must be built for the target architecture. A pre-built arm64 binary is not universally available.

4. **Verilator** – https://www.veripool.org/verilator/

   Handles Verilog simulation and must be installed separately.

5. **XQuartz** – https://www.xquartz.org

   provides the X11 windowing system required by ngspice for interactive plot windows. It is not installed by default on macOS, and the DISPLAY environment variable it relies on is not inherited by .app bundles launched from Finder.

The Python package dependencies — PyQt5, numpy, matplotlib, scipy, watchdog, psutil, and hdlparse — are installed into the virtual environment and bundled inside the .app by PyInstaller. eSim source is licensed under the GNU General Public License.

2.5 Prior State of MacOS Support

Prior to this project, eSim had no macOS support of any kind. Its installation scripts are written entirely for Linux, using apt package manager calls and Linux-specific paths throughout. The source code contained several implicit Linux assumptions: tools were invoked by bare command names not present in the macOS PATH, Validation.py used shutil.which() which cannot locate binaries inside macOS application bundles, and environment variables such as SPICE_SCRIPTS and DISPLAY were assumed to be set in the shell, which is true in a Linux terminal but not for a .app launched from Finder.

A macOS user wishing to run eSim would need to manually compile ngspice, install KiCad and XQuartz, configure a Python virtual environment, set shell environment variables, and launch eSim from the terminal by activating the environment and running python3 Application.py from the correct working directory. This requires significant technical knowledge, is time-consuming, and produces a fragile machine-specific installation entirely unsuitable for general distribution.

This project addresses each of these issues systematically, resulting in a self-contained .app bundle and a .pkg installer that reduces installation to a single double-click.

Chapter 3

System Design and Architecture

3.1 System Overview

This project focuses on designing a macOS-compatible system for eSim by integrating its components into a unified and easily deployable structure. The system is designed to bundle all required dependencies, ensuring that users can run eSim without manual configuration or external setup. It aims to reduce the complexity involved in installing and configuring multiple tools by providing a streamlined and automated installation process. The overall design emphasizes portability, ease of installation, and compatibility across different macOS architectures, including both Intel and Apple Silicon systems.

All development and testing for this project was carried out on a single machine running Apple Silicon macOS. The system specifications are shown in Table 3.1.

Property	**Value**
Machine	MacBook Pro (Apple Silicon)
macOS Version	macOS 26.0.1 (Tahoe)
Chip Architecture	arm64 (Apple Silicon M-series)
Build Version	25A362

Table 3.1: Development Machine Specifications

3.2 Tools and Versions Installed

Before any eSim-specific work could begin, the following tools were installed and verified on the development machine. Table 3.2 lists each tool, its version, and the method of installation.

In the following tools, Qt binaries are not added to PATH by default after installation. The following line was added to ∼/.zshrc to make qmake and macdeployqt accessible from the terminal:

```
export PATH=$HOME/Qt/6.10.1/macos/bin:$PATH
```

Tool	Version	Installation Method
Xcode / Clang	Apple Clang 17.0.0	Xcode App Store
CMake	4.2.0	Homebrew
Ninja	1.13.2	Homebrew
Git	2.53.0	Homebrew
Qt	6.10.1	Qt Online Installer
qmake	3.1	Included with Qt 6.10.1
macdeployqt	—	Included with Qt 6.10.1
XQuartz	2.8.x	xquartz.org
KiCad	6.0	kicad.org
Python	3.x (system)	macOS built-in

Table 3.2: Installed Tools

3.3 Python Virtual Environment

A dedicated Python virtual environment named `esim-env` was created inside the eSim project directory to isolate all Python dependencies from the system Python installation. This ensures that the packages bundled by PyInstaller are exactly those required by eSim, with no contamination from system-wide packages.

Listing 3.1: Ngspice Build Script for Apple Silicon

```
# Create the virtual environment
python3 -m venv esim-env

# Activate it
source esim-env/bin/activate

# Install eSim's Python dependencies
pip install PyQt5 numpy matplotlib scipy watchdog psutil
   pyinstaller makerchip-app sandpiper-saas setuptools

# Install hdlparse from source (Python 3 compatible fork)
pip install https://github.com/aishw31/hdlparse/tarball/
   python3compat
```

The virtual environment must be activated before running eSim in development or before invoking PyInstaller. All subsequent commands in this report assume the environment is active.

3.4 Running eSim in Development

Before any packaging work began, eSim was verified to run correctly in the development environment. The standard launch sequence is:

Listing 3.2: Standard Launch Sequence

```
# Navigate to the frontEnd source directory
cd ~/Projects/eSim-2.5/src/frontEnd

# Activate the virtual environment
source ../../esim-env/bin/activate

# Set required environment variables
export SPICE_SCRIPTS=/usr/local/share/ngspice/scripts
export DISPLAY=:0

# Launch the application
python3 Application.py
```

A key observation at this stage was that the SPICE_SCRIPTS and DISPLAY environment variables had to be set manually in the terminal for ngspice simulation and interactive plot windows to work. These variables are present in the shell environment during development.

3.5 The macOS Application Bundle Structure

To create a native macOS experience, the bundled application must adhere to Apple's uniform `.app` directory structure. Unlike Linux, where executables and resources are scattered across `/usr/bin` and `/usr/share`, a macOS application bundle encapsulates all necessary data.

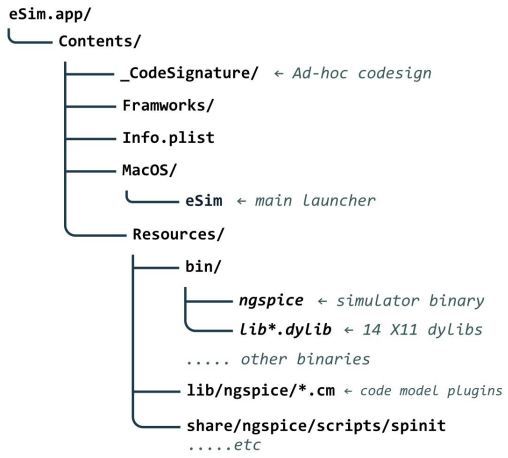

Figure 3.1: Typical MacOS .app Bundle Structure

Chapter 4

Ngspice Build & Configuration

4.1 Overview

Ngspice is the SPICE simulation engine that eSim uses to run circuit simulations. On Linux, ngspice is available through the system package manager and installs cleanly with all required components. On macOS, no pre-built package exists that satisfies all of eSim's requirements, making a manual build from source necessary. The version used throughout this project is ngspice 35. This version was already validated against eSim's Python simulation pipeline and plotting infrastructure, making it the appropriate choice for stability. The build process on macOS introduced several challenges that are documented in this chapter, along with the solutions applied to each.

4.2 Build Challenges

Before arriving at a working build, several issues were encountered. These are summarised in Table 4.1 and described in detail in the sections that follow.

Issue	**Description**
X11 dependencies	Existing installation linked against `/opt/X11/*` not portable without XQuartz at that exact path
OpenMP unavailable	`omp.h` not present on macOS by default; standard configure script fails
spinit relative paths	Initialisation script used `../lib/ngspice/` paths that break outside the install directory
Code model plugins missing	`.cm` files not found at runtime on fresh machines
SPICE_SCRIPTS not inherited	Environment variable not passed to `.app` bundles launched from Finder

Table 4.1: ngspice Build Challenges on macOS

4.3 Configure Flags and Build Strategy

The final working build of ngspice 35 uses the following configure invocation:

```
./configure \
  --enable-xspice \
  --enable-cider \
  --disable-openmp \
  --disable-debug \
  --enable-relpath \
  --without-readline \
  CFLAGS= -m64 -O2 \
    -I/opt/X11/include \
    -I/opt/X11/include/freetype2 \
    -I/opt/homebrew/Cellar/freetype/2.14.2/include/freetype2 \
    -I/opt/homebrew/Cellar/freetype/2.14.2/include  \
  LDFLAGS= -m64 -L/opt/X11/lib
```

The key flags and their rationale and some important ones are explained in the following

`--enable-xspice` Enables the XSPICE event-driven simulation extensions, required for eSim's code model support

`--enable-cider` Enables the CIDER mixed-level simulator for device-level simulation

`--disable-openmp` OpenMP (omp.h) is not available on macOS without manual installation; disabling avoids the build failure

`--without-readline` Disables readline support; not required for eSim's subprocess-based invocation of ngspice

After configure completes successfully, the build and installation are run as:

```
$ make -j$(sysctl -n hw.ncpu)
$ sudo make install
```

The `-j$(sysctl -n hw.ncpu)` flag parallelises the build across all available CPU cores, significantly reducing build time on Apple Silicon. The resulting binary is installed to `/usr/local/bin/ngspice`. Its architecture can be verified with:

```
$ file /usr/local/bin/ngspice
$ /usr/local/bin/ngspice: Mach-O 64-bit executable arm64
```

Chapter 5

macOS Source Code Modifications

5.1 Overview

The reference taken to the eSim source code is entirely based on the Linux version. Several files contained implicit assumptions about the operating environment that do not hold on macOS. This chapter documents some file that required modification, the problem it presented on macOS, and the fix applied. A summary of all modified files is given in Table **??**.

File	Module	Nature of Change
Kicad.py	projManagement	macOS eeschema path
Validation.py	projManagement	macOS tool detection
NgspiceWidget.py	ngspiceSimulation	Binary path, DISPLAY, SPICE_SCRIPTS
macOSSetup.py	frontEnd	KiCad symbol library registration
pathmagic.py	frontEnd	Working directory resolution
Application.py	frontEnd	Startup environment setup

Table 5.1: Source Files Modified for macOS

5.2 Kicad.py — eeschema Path Resolution

eSim launches KiCad's schematic editor (`eeschema`) as an external subprocess. On Linux, `eeschema` is a standalone command available in `PATH`. On macOS, it is embedded inside the KiCad application bundle and is not accessible as a bare command. The original invocation in `Kicad.py` was:

Listing 5.1: Original eeschema invocation (Linux only)

```
1 self.cmd = 'eeschema ' + schematic_file
```

This fails silently on macOS because `eeschema` does not exist in `PATH`. The fix adds a `sys.platform` check that constructs the full path to the binary inside the KiCad application bundle:

Listing 5.2: macOS-specific eeschema path fix

```
import sys

if os.environ.get('ESIM_FLATPAK') == '1':
    self.cmd = ('flatpak␣run␣--command=eeschema␣'
                'org.kicad.KiCad␣' + schematic_file)

elif sys.platform == 'darwin':
    eeschema_path = (
        '/Applications/KiCad/KiCad.app/Contents/Applications'
        '/eeschema.app/Contents/MacOS/eeschema'
    )
    self.cmd = eeschema_path + '␣' + schematic_file

else:
    self.cmd = 'eeschema␣' + schematic_file
```

The `elif sys.platform == 'darwin'` branch is new. The Flatpak branch was already present for Linux; the `else` branch covers standard Linux installations. The macOS branch constructs the absolute path to `eeschema` within KiCad 6's standard installation location.

5.3 Validation.py — Tool Detection

`Validation.py` contains a `validateTool()` method that checks whether required external tools are present before eSim attempts to use them. The original implementation used `shutil.which()` exclusively:

Listing 5.3: Original validateTool — PATH only

```
def validateTool(self, toolName):
    if shutil.which(toolName) is not None:
        return True

    return False
```

`shutil.which()` searches only `PATH`. On macOS, tools such as `eeschema` live inside application bundles and are not in `PATH`, so this check always returns `False`, causing eSim to report missing tools even when they are correctly installed. The fix extends the method with the following macOS-specific path check:

Listing 5.4: Extended validateTool with macOS bundle path check

```
def validateTool(self, toolName):
    import sys
    if shutil.which(toolName) is not None:
        return True
    if sys.platform == 'darwin':
        mac_paths = {
            'eeschema': (
                '/Applications/KiCad/KiCad.app/Contents/
                   Applications'
                '/eeschema.app/Contents/MacOS/eeschema'
            ),
            'ngspice': '/usr/local/bin/ngspice',
        }
        path = mac_paths.get(toolName)
        if path and os.path.isfile(path):
            return True
    return False
```

5.4 NgspiceWidget.py — Binary Path, DISPLAY, and SPICE_SCRIPTS

`NgspiceWidget.py` manages the ngspice simulation process and its interactive plot windows. Three changes were required.

5.4.1 Ngspice Binary Path

The original code invoked ngspice by its bare name, relying on `PATH`:

Listing 5.5: Original ngspice invocation

```
self.process.start('ngspice', args)
```

On macOS the binary is at `/usr/local/bin/ngspice`, which may not be in the `PATH` of a `.app` launched from Finder. The fix uses a helper that returns the correct path in both development and bundled contexts:

Listing 5.6: Updated ngspice binary path resolution

```
def _get_ngspice_binary(self):
    bundled = get_bundle_bin('ngspice')
    if os.path.isfile(bundled):
        return bundled
    return '/usr/local/bin/ngspice'

# Usage:
self.process.start(self._get_ngspice_binary(), args)
```

5.4.2 SPICE_SCRIPTS Resolution

Similarly, the path to ngspice's `scripts/` directory is resolved through a helper that checks the bundle first:

Listing 5.7: SPICE_SCRIPTS path resolution

```
def _get_spice_scripts(self):
    bundled = get_bundle_share('ngspice/scripts')
    if os.path.isdir(bundled):
        return bundled
    return '/usr/local/share/ngspice/scripts'
```

5.4.3 DISPLAY Resolution — _ensure_display()

Ngspice's interactive plot windows require an X11 display connection, indicated by the `DISPLAY` environment variable. A `.app` launched from Finder does not inherit shell environment variables, so `DISPLAY` is typically unset, causing ngspice to fail:

Listing 5.8: Error when DISPLAY is unset

```
ERROR: This operation is not defined for display type PrinterOnly.
Can`t open viewport for graphics.
```

`_ensure_display()` method was added to resolve `DISPLAY` through a priority chain:

Listing 5.9: _ensure_display() implementation

```
def _ensure_display(self):
    import glob, subprocess
    # 1. Already set
    if os.environ.get('DISPLAY'):
        return os.environ['DISPLAY']

    # 2. Query launchctl (populated when XQuartz has run before)
    try:
        result = subprocess.run(
            ['launchctl', 'getenv', 'DISPLAY'],
            capture_output=True, text=True, timeout=3
        )
        if result.stdout.strip():
            os.environ['DISPLAY'] = result.stdout.strip()
            return os.environ['DISPLAY']
    except Exception:
        pass

    # 3. Search for XQuartz Unix socket directly
    sockets = glob.glob(
        '/private/tmp/com.apple.launchd.*/org.xquartz:0'
    )
    if sockets:
        os.environ['DISPLAY'] = sockets[0]
        return os.environ['DISPLAY']
```

```
26
27      # 4. Auto-start XQuartz and poll for socket
28      try:
29          subprocess.Popen(['open', '-a', 'XQuartz'])
30          import time
31          for _ in range(16):
32              time.sleep(0.5)
33              sockets = glob.glob(
34                  '/private/tmp/com.apple.launchd.*/org.xquartz:0'
35              )
36              if sockets:
37                  os.environ['DISPLAY'] = sockets[0]
38                  return os.environ['DISPLAY']
39      except Exception:
40          pass
41
42      # 5. Hardcoded fallback
43      os.environ['DISPLAY'] = ':0'
44      return ':0'
```

Table 5.2 summarises the priority chain.

Table 5.2: `_ensure_display()` Priority Chain

Priority	Method	Description
1	`os.environ` check	Use if already set
2	`launchctl getenv`	Populated when XQuartz has run at least once
3	Socket glob	Search `/private/tmp/com.apple.launchd.*` directly
4	Auto-start XQuartz	Launch XQuartz via `open -a`, poll 8 seconds
5	Fallback `:0`	Last resort hardcoded default

5.5 macOSSetup.py — KiCad Symbol Library Registration

`macOSSetup.py` registers eSim's KiCad symbol libraries by writing entries into KiCad's `sym-lib-table` file. The original implementation assumed source-relative paths that break inside a frozen `.app`. The following fixes were applied:

1. **Path resolution** — `get_esim_symbols_dir()` was rewritten to check `sys._MEIPASS` when `sys.frozen` is `True`, before falling back to source-relative paths.

2. **Stale flag detection** — The registration flag file is now re-validated on every launch. If the flag exists but the symbols path no longer resolves, the flag is cleared and registration re-runs.

3. **Missing table creation** — `ensure_sym_lib_table_exists()` was added to create a minimal `sym-lib-table` if KiCad was installed but never opened.

Listing 5.10: Frozen-aware symbol directory resolution

```
def get_esim_symbols_dir():
    if getattr(sys, 'frozen', False):
        # Running inside PyInstaller .app
        contents_dir = os.path.dirname(os.path.dirname(sys.
           executable))
        candidates = [
            os.path.join(sys._MEIPASS, 'library', 'ngspiceLibrary')
               ,
            os.path.join(contents_dir, 'Resources',
                         'library', 'ngspiceLibrary'),
        ]
    else:
        base = os.path.abspath(
            os.path.join(os.path.dirname(__file__), '..', '..')
        )
        candidates = [
            os.path.join(base, 'library', 'ngspiceLibrary'),
        ]

    for path in candidates:
        if os.path.isdir(path):
            return path

    return None
```

5.6 pathmagic.py: Working Directory Resolution

`pathmagic.py` sets up Python's import path by using `os.getcwd()`, which assumes eSim is launched from the `src/frontEnd/` directory. This holds in a terminal session but not when launching a `.app` from Finder, where the working directory is `/`. For the development environment the fix is to always launch from the correct directory:

Listing 5.11: Correct launch directory for development

```
cd ~/Projects/eSim-2.5/src/frontEnd
source ../../esim-env/bin/activate
python3 Application.py
```

For the `.app` bundle, the PyInstaller launcher script sets `PYTHONPATH` correctly so that `pathmagic.py` finds the right base path without depending on the working directory.

5.7 Application.py — Startup Environment Setup

`Application.py` is the entry point of the application. Two additions were made for macOS. First, `SPICE_SCRIPTS` is set programmatically before any ngspice subprocess is invoked:

Listing 5.12: SPICE_SCRIPTS set at startup

```
import os

if 'SPICE_SCRIPTS' not in os.environ:
    os.environ['SPICE_SCRIPTS'] = '/usr/local/share/ngspice/scripts
        '
```

Second, all image and icon paths that previously used the `init_path` pattern were updated to use `resource_path()` from `esim_paths.py` (described in Chapter 6):

Listing 5.13: Image path update using resource_path()

```
# Before:
init_path = os.path.abspath(
    os.path.join(os.path.dirname(__file__), '..', '..')
)
self.setWindowIcon(QtGui.QIcon(init_path + '/images/logo.png'))

# After:
from esim_paths import resource_path
self.setWindowIcon(QtGui.QIcon(resource_path('images', 'logo.png'))
    )
```

Chapter 6

PyInstaller Bundling

6.1 Overview

PyInstaller works by statically analysing a Python application's imports, collecting all required modules, assets, and a Python interpreter, and producing a self-contained `.app` directory that runs without a Python installation on the target machine. For a `--onedir` build on macOS, all collected files are placed in `Contents/MacOS/` inside the `.app` bundle.

The initial PyInstaller build of eSim produced a `.app` that launched but exhibited three failures immediately. This chapter documents those failures, their root cause, and all fixes applied to produce a fully working bundle.

6.2 Initial Build

The first build was produced using the following command from the project root, with `esim-env` activated:

Listing 6.1: Initial PyInstaller build command

```
pyinstaller \
  --name eSim \
  --windowed \
  --onedir \
  --add-data  images:images  \
  --add-data  library:library  \
  --add-data  src:src  \
  --add-data  Examples:Examples  \
  --add-data  nghdl:nghdl  \
  src/frontEnd/Application.py
```

The resulting `.app` was launched directly from the terminal to capture stdout and stderr:

Listing 6.2: Launching the .app from terminal for debugging

```
./dist/eSim.app/Contents/MacOS/eSim
```

The terminal output revealed three errors:

Listing 6.3: Errors on first .app launch

```
macOS KiCad setup warning: No module named 'macOSSetup'
QTextBrowser: No document for file:///...eSim.app/Contents/MacOS/
    library/browser/welcome.html
QPixmap::scaled: Pixmap is a null pixmap
```

Table 6.1 summarises the three errors and their causes.

Table 6.1: Errors on First .app Launch

Error	**Root Cause**	**Fix**
`No module named 'macOSSetup'`	Dynamic import not detected by PyInstaller's static analysis	`hiddenimports` + `pathex` in `.spec`
`welcome.html not found`	`__file__` resolves to binary in frozen app; path navigation breaks	`resource_path()` helper + `sys._MEIPASS`
`QPixmap is a null pixmap`	Same `__file__` issue; images directory not found	`resource_path()` across all asset-loading files

6.3 The __file__ Problem in Frozen Apps

The most pervasive issue in PyInstaller bundling of complex Python applications is the behaviour of the `__file__` built-in variable in a frozen context. In development, `__file__` inside any source file resolves to that file's absolute path on disk. Code that navigates to the project root using `os.path.dirname(__file__)` and relative traversal works correctly:

Listing 6.4: Path navigation in development — works correctly

```
# In src/frontEnd/Application.py during development:
# __file__ = /Users/.../eSim-2.5/src/frontEnd/Application.py

init_path = os.path.abspath(
    os.path.join(os.path.dirname(__file__), '..', '..')
)
# init_path = /Users/.../eSim-2.5/
# init_path + '/images/logo.png' exists
```

Inside a frozen `.app`, `__file__` resolves to the compiled binary executable, not any source file. The same navigation then produces an incorrect result:

Listing 6.5: Path navigation inside frozen .app — breaks

```
# In frozen .app:
# __file__ = .../eSim.app/Contents/MacOS/eSim  (the binary)

init_path = os.path.abspath(
    os.path.join(os.path.dirname(__file__), '..', '..')
)
# init_path = .../eSim.app/
# init_path + '/images/logo.png' does not exist
```

PyInstaller unpacks all bundled data files into a temporary directory accessible at runtime via `sys._MEIPASS`. In a `--onedir` build this corresponds to `Contents/MacOS/`. All `datas` entries — `images/`, `library/`, `src/`, etc. — are placed there. The correct base path is therefore `sys._MEIPASS`, not a path derived from `__file__`.

6.4 Fix: esim_paths.py

A shared helper module, `src/frontEnd/esim_paths.py`, was created to centralise frozen/development path detection. All files that need to resolve top-level project assets import from this module rather than duplicating the detection logic.

Listing 6.6: esim_paths.py — shared path resolution helper

```
# src/frontEnd/esim_paths.py

import os
import sys

def get_base_dir():

    Returns the project root directory.
    - Frozen PyInstaller .app: sys._MEIPASS (Contents/MacOS/)
    - Development: two levels up from this file
    """
    if getattr(sys, 'frozen', False):
        return sys._MEIPASS
    return os.path.abspath(
        os.path.join(os.path.dirname(__file__), '..', '..')
    )

def resource_path(*relative_parts):
       Build an absolute path to a bundled resource.
    return os.path.join(get_base_dir(), *relative_parts)
```

Usage and modifications across all codebase follows the same pattern. Table 6.2 identifies which `__file__` usage patterns are safe and which require updating.

Table 6.2: __file__ Path Patterns — Safe vs Broken in Frozen App

Pattern	Status	Action
`__file__` → sibling file (same dir)	Safe	None
`__file__` → subdirectory within package	Safe	None
`__file__` + `../../` → project root	Broken	Use `resource_path()`
`__file__` + `../../images/`	Broken	Use `resource_path()`
`__file__` + `../../library/`	Broken	Use `resource_path()`

6.5 .spec File Configuration

The auto-generated `Application.spec` file was modified to declare all required assets, hidden imports, and analysis paths. The final `.spec` is shown below.

Listing 6.7: Final Application.spec

```
a = Analysis(
    ['src/frontEnd/Application.py'],
    pathex=['src/frontEnd'], # Allows macOSSetup to be found
    binaries=[],
    datas=[
        ('images',   'images'),
        ('library',  'library'),
        ('Examples', 'Examples'),
        ('src',      'src'),
        ('nghdl',    'nghdl'),
    ],
    hiddenimports=['macOSSetup', 'PyQt5.sip'],
    // other default settings
)
pyz = PYZ(a.pure)
exe = EXE(
    name='eSim',
    // other default settings
)
coll = COLLECT(
    // other default settings
    name='eSim',
)
app = BUNDLE(
    coll,
    name='eSim.app', #eSim App Name
    icon='eSimIcon.icns', #eSim Icon
    bundle_identifier='esim.fossee.in',
)
```

The two additions relative to the auto-generated spec are:

- `pathex=['src/frontEnd']` — adds the `frontEnd` directory to PyInstaller's analysis path so that `macOSSetup.py` is found as a proper Python module.
- `hiddenimports=['macOSSetup', 'PyQt5.sip']` — explicitly declares modules that are imported dynamically at runtime and are therefore invisible to PyInstaller's static analysis.

6.6 Rebuild Workflow

After any change to the source code or `.spec` file, the build should always be performed using the `.spec` file directly rather than re-running `pyinstaller` with `--add-data` flags, as it regenerates the `.spec` and overwrites all manual changes.

Listing 6.8: Correct rebuild workflow

```
# 1. Remove stale build artefacts
rm -rf dist/ build/

# 2. Activate the virtual environment
source esim-env/bin/activate

# 3. Build using the spec file
pyinstaller Application.spec

# 4. Launch from terminal to inspect stdout/stderr
./dist/eSim.app/Contents/MacOS/eSim
```

Chapter 7

X11/XQuartz Integration

7.1 Overview

Ngspice uses X11 for its interactive graphical plot windows. On macOS, X11 is provided by XQuartz, a separately installed compatibility layer. Integrating ngspice's X11 dependency into the `.app` bundle required solving two independent problems: ensuring the `DISPLAY` environment variable is correctly set at runtime, and bundling the XQuartz shared libraries that ngspice links against so that the `.app` works on machines where XQuartz is not installed at the expected path. This chapter focuses on the shared library bundling and relinking strategy.

7.2 X11 Dylib Dependency Chain

Running `otool -L` on the built ngspice binary reveals shared library dependencies:

Listing 7.1: otool -L output for ngspice binary (X11 dependencies)

```
$ otool -L /usr/local/bin/ngspice
/usr/local/bin/ngspice:
    /opt/X11/lib/libXaw.7.dylib
    /opt/X11/lib/libXmu.6.dylib
    /opt/X11/lib/libXt.6.dylib
    /opt/X11/lib/libXext.6.dylib
    /opt/X11/lib/libX11.6.dylib
    /opt/X11/lib/libXft.2.dylib
    /opt/X11/lib/libfontconfig.1.dylib
    /opt/X11/lib/libXrender.1.dylib
    /opt/X11/lib/libfreetype.6.dylib
    /opt/X11/lib/libSM.6.dylib
    /opt/X11/lib/libICE.6.dylib
    /usr/lib/libSystem.B.dylib
    /usr/lib/libc++.1.dylib
```

The `/opt/X11/lib/` entries are XQuartz libraries. This path does not exist on macOS systems without XQuartz installed, meaning the `.app` would fail to launch ngspice on any such machine. All eleven dylibs must be copied into the bundle and the binary relinked to find them there.

Table 7.1 lists the libraries and their roles.

Table 7.1: XQuartz Dylibs Bundled in the .app

Library	**Role**
`libXaw.7.dylib`	Athena widget set (ngspice plot UI)
`libXmu.6.dylib`	Miscellaneous X utilities
`libXt.6.dylib`	X Toolkit intrinsics
`libXext.6.dylib`	X11 extensions
`libX11.6.dylib`	Core X11 client library
`libXft.2.dylib`	X FreeType font rendering
`libfontconfig.1.dylib`	Font configuration
`libXrender.1.dylib`	X Render extension
— 6 more libraries —	— 6 more libraries —

7.3 Relinking Strategy

When a macOS binary loads a shared library, it uses the install name recorded in the binary at link time. The install names in the ngspice binary are absolute paths beginning with `/opt/X11/lib/`. To make the binary load the bundled copies instead, these install names must be replaced with paths relative to the binary's location using the `@loader_path` prefix.

7.4 relink_ngspice.sh

A shell script, `relink_ngspice.sh`, was written to perform the relinking automatically after each PyInstaller build. It carries out three operations:

1. Copies all ngspice dylibs from `/opt/X11/lib/` into `Contents/Resources/bin/` bundled directory.
2. Patches the ngspice binary, replacing each `/opt/X11/lib/libX.dylib` reference with `@loader_path/libX.dylib`.
3. Patches each dylib's own install name and internal references to other X11 dylibs.
4. Ad-hoc codesigns all patched binaries, required on Apple Silicon.

Listing 7.2: relink_ngspice.sh

```
#!/bin/bash
set -e

APP=dist/eSim.app
BIN=$APP/Contents/Resources/bin
NGSPICE=$BIN/ngspice
X11=/opt/X11/lib

DYLIBS=(
    libXaw.7.dylib libXmu.6.dylib libXt.6.dylib
    libXext.6.dylib libX11.6.dylib libXft.2.dylib
    libfontconfig.1.dylib libXrender.1.dylib
    libfreetype.6.dylib libSM.6.dylib libICE.6.dylib
)

# 1. Copy dylibs into the bundle
mkdir -p  $BIN
for lib in  ${DYLIBS[@]} ; do
    cp  $X11/$lib   $BIN/
done

# 2. Patch ngspice binary
for lib in  ${DYLIBS[@]} ; do
    install_name_tool \
        -change  $X11/$lib   @loader_path/$lib  \
         $NGSPICE
done

# 3. Patch each dylib's own install name and cross-references
for lib in  ${DYLIBS[@]} ; do
    install_name_tool \
        -id  @loader_path/$lib  \
         $BIN/$lib
    for dep in  ${DYLIBS[@]} ; do
        install_name_tool \
            -change  $X11/$dep   @loader_path/$dep  \
             $BIN/$lib  2>/dev/null || true
    done
done

# 4. Ad-hoc codesign all patched binaries (required on Apple
   Silicon)
codesign --force --sign -  $NGSPICE
for lib in  ${DYLIBS[@]} ; do
    codesign --force --sign -  $BIN/$lib
done
echo  Relinking␣complete.
```

The script is run once after each PyInstaller build:

Listing 7.3: Running the relink script after build

```
pyinstaller Application.spec
bash relink_ngspice.sh
```

Chapter 8

ngspice Code Model Plugins

8.1 Overview

Ngspice's XSPICE extensions rely on a set of model plugins, with the `.cm` extension. These plugins are loaded at runtime when ngspice processes a `codemodel` directive in its `spinit` initialisation script. On a standard system installation, these files reside at `/usr/local/lib/ngspice/`. When ngspice is bundled inside the `.app`, this system path does not exist causing every code model to fail to load.

8.2 Problem: Missing .cm Files

On the target machine without a local ngspice installation, launching eSim and running a simulation produced the following errors in the simulation console:

Listing 8.1: Code model loading errors on fresh machine

```
# Error: Library /usr/local/lib/ngspice/spice2poly.cm couldn't be loaded!
# Error opening code model: dlopen(...) no such file or directory
# Error: Library /usr/local/lib/ngspice/analog.cm couldn't be loaded!
```

The root cause is the hardcoded absolute path `/usr/local/lib/ngspice/` in the `spinit` script. Even after the relative-path fix, the absolute path works only on the development machine where ngspice is system-installed.

8.3 Bundling .cm Files

The `.cm` files are added to the `datas` list in the `.spec` file, mapping the system `lib/ngspice/` directory to a `lib/ngspice/` directory inside the bundle:

Listing 8.2: Adding .cm files to the .spec datas list

```
datas=[
    # other inclued datas
    ('/usr/local/lib/ngspice', 'lib/ngspice'), # .cm plugins
    ('/usr/local/share/ngspice', 'share/ngspice'), # spinit scripts
],
```

8.4 _generate_spinit(): Runtime Path Resolution

Simply bundling the `.cm` files is not sufficient. The `spinit` script still contains the hardcoded absolute path `/usr/local/lib/ngspice/`. This path must be replaced at runtime with the actual path to the bundled `lib/ngspice/` directory inside the `.app`.

A method `_generate_spinit()` was added to `NgspiceWidget.py`. It reads the bundled `spinit` template, replaces the hardcoded path with the correct bundle-relative path, writes a resolved copy to a temporary directory, and sets `SPICE_SCRIPTS` to point to that directory before starting the ngspice process.

Listing 8.3: _generate_spinit() implementation

```
import tempfile, shutil

def _generate_spinit(self):
    # Path to bundled spinit template
    bundled_scripts = self._get_spice_scripts()
    spinit_template = os.path.join(bundled_scripts, 'spinit')

    # Path to bundled .cm files
    if getattr(sys, 'frozen', False):
        contents_dir = os.path.dirname(os.path.dirname(sys.
            executable))
        cm_dir = os.path.join(
            contents_dir, 'Resources', 'lib', 'ngspice'
        )
    else:
        cm_dir = '/usr/local/lib/ngspice'

    # Read template, replace hardcoded path
    with open(spinit_template, 'r') as f:
        content = f.read()

    content = content.replace(
        '/usr/local/lib/ngspice', cm_dir
    )

    # Write resolved spinit to a temp directory
    tmp_dir = tempfile.mkdtemp(prefix='esim_spice_')
    resolved_spinit = os.path.join(tmp_dir, 'spinit')
    with open(resolved_spinit, 'w') as f:
        f.write(content)

    # Set SPICE_SCRIPTS to temp dir so ngspice finds it
    os.environ['SPICE_SCRIPTS'] = tmp_dir
    return tmp_dir
```

This method is called in `NgspiceWidget` before the ngspice `QProcess` is started, ensuring that every simulation session uses a correctly resolved `spinit`.

Chapter 9

Final .app Bundle Structure

9.1 Directory Layout

The complete `eSim.app` bundle produced after PyInstaller and the relinking script has the following directory structure:

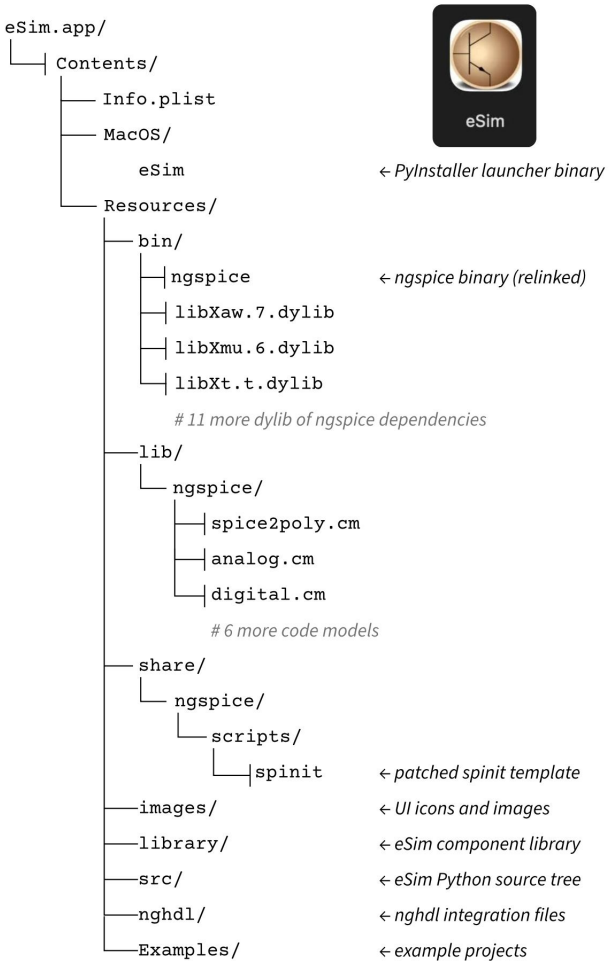

Figure 9.1: eSim.app directory layout

9.2 Runtime Path Resolution Summary

Table 9.1 summarises how each major resource is located at runtime, both in the development environment and inside the frozen `.app`.

Table 9.1: Runtime Path Resolution — Development vs Frozen .app

Resource	**Development Path**	**Frozen .app Path**
Project root	`src/frontEnd/../../`	`sys._MEIPASS`
Images	`<root>/images/`	`sys._MEIPASS/images/`
welcome.html	`<root>/library/browser/`	`sys._MEIPASS/library/browser/`
ngspice binary	`/usr/local/bin/ngspice`	`Contents/Resources/bin/ngspice`
.cm plugins	`/usr/local/lib/ngspice/`	`Contents/Resources/lib/ngspice/`
spinit	`/usr/local/share/ngspice/scripts/`	Resolved copy in `tmpdir/` via `_generate_spinit()`
KiCad symbols	Source-relative path	`sys._MEIPASS` or `Contents/Resources/`

Chapter 10

macOS .pkg Installer

10.1 Overview

The `.app` bundle produced in the previous chapters is self-contained, but distributing it as a bare `.app` requires the user to manually copy it to `/Applications` and handle initial configuration. A `.pkg` installer automates this — placing the application, configuring the environment, and setting permissions through a standard macOS installation wizard. The installer is built using `pkgbuild`, Apple's command-line tool for creating `.pkg` packages, which supports two hook scripts: `preinstall` and `postinstall`, running before and after file installation respectively.

10.2 pkgbuild Command

The final `pkgbuild` invocation is:

```
pkgbuild \
  --root dist/eSim.app \
  --install-location /Applications/eSim.app \
  --scripts scripts/ \
  --identifier esim.fossee.in \
  --version 2.5 \
  eSim.pkg
```

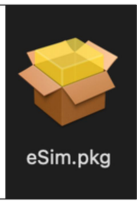

Figure 10.1: pkgbuild command for eSim.pkg

Table 10.1 explains each flag.

Table 10.1: pkgbuild Flag Reference

Flag	**Purpose**
`--root dist/eSim.app`	Source directory whose contents are installed
`--install-location`	Destination path on the target machine
`--scripts scripts/`	Directory containing `preinstall` and `postinstall`
`--identifier esim.fossee.in`	Reverse-DNS bundle identifier for the package
`--version 2.5`	Package version string

10.3 preinstall Script

The `preinstall` script runs before any files are copied to the target machine. Its responsibility is to detect required dependencies and ensure they are present before eSim is installed.

Listing 10.1: preinstall script

```
#!/bin/bash

set -e
echo "eSim preinstall: checking dependencies..."

# Check KiCad 6
if [ ! -d "/Applications/KiCad/KiCad.app" ]; then
    echo "Warning: KiCad 6 not found at /Applications/KiCad."
fi

# Check XQuartz
if [ ! -d "/Applications/Utilities/XQuartz.app" ]; then
    echo "Warning: XQuartz not found."
    echo "ngspice interactive plots require XQuartz."
fi

# Check Rosetta 2 (Apple Silicon only)
ARCH=$(uname -m)
if [ "$ARCH" = "arm64" ]; then
    if ! /usr/bin/pgrep -q oahd; then
        echo "Installing Rosetta 2..."
        /usr/sbin/softwareupdate --install-rosetta --agree-to-
            license
    fi
fi

# Check Xcode Command Line Tools
if ! xcode-select --print-path &>/dev/null; then
    echo "Warning: Xcode Command Line Tools not found."
fi

echo "preinstall checks complete."
exit 0
```

10.4 postinstall Script

The `postinstall` script runs after the `.app` has been placed in `/Applications`. It handles three tasks: creating configuration directories, removing the macOS quarantine attribute, and setting execute permissions.

Listing 10.2: postinstall script

```
#!/bin/bash
set -e
echo  eSim postinstall: configuring...
# 1. Create configuration directories
mkdir -p  $HOME/.esim
mkdir -p  $HOME/.nghdl
# Write default eSim configuration if not already present
ESIM_CONFIG= $HOME/.esim/config
if [ ! -f  $ESIM_CONFIG  ]; then
    cat >  $ESIM_CONFIG  << EOF
[eSim]
workspace=$HOME/eSim-Workspace
EOF
fi
echo  Configuration directories created.
# 2. Remove macOS quarantine attribute
# Without this, Gatekeeper blocks the bundled ngspice binary
xattr -rd com.apple.quarantine /Applications/eSim.app
echo  Quarantine attribute removed.
# 3. Set execute permissions on binaries and scripts
chmod +x /Applications/eSim.app/Contents/MacOS/eSim
chmod +x /Applications/eSim.app/Contents/Resources/bin/ngspice
echo  Permissions set. eSim installation complete.
exit 0
```

10.5 Configuration Directories

The two configuration directories created by the `postinstall` script are:

Table 10.2: Configuration Directories Created by postinstall

Directory	Used By	Contents
`~/.esim/`	eSim	Workspace path, user preferences, project history
`~/.nghdl/`	nghdl module	GHDL bridge configuration

10.6 Quarantine Removal

macOS assigns a quarantine extended attribute (`com.apple.quarantine`) to any file downloaded from the internet, and Gatekeeper will block unsigned binaries carrying it with a "cannot be opened" dialog. The `postinstall` script removes that attribute recursively using:

Listing 10.3: Quarantine removal command

```
xattr -rd com.apple.quarantine /Applications/eSim.app
```

Chapter 11

Testing & Validation

11.1 Test Environment

All testing was performed on the development machine, running macOS 26.0.1 on Apple Silicon. Testing covered four areas: `.app` launch and UI correctness, ngspice simulation, KiCad integration, and installer behaviour.

11.2 .app Launch Verification

The `.app` was launched both from the terminal (to capture stdout and stderr) and from Finder (to simulate end-user conditions). Table 11.1 summarises the checks performed.

Table 11.1: .app Launch Verification Results

Check	**Method**	**Result**
Main window appears	Visual	Pass
Window icon loads	Visual	Pass
welcome.html renders	Visual	Pass
No `QPixmap` errors	Terminal	Pass
No module import errors	Terminal	Pass
macOSSetup runs without error	Terminal	Pass
Finder launch (no terminal)	Visual	Pass

Figure 11.1: Launched eSim

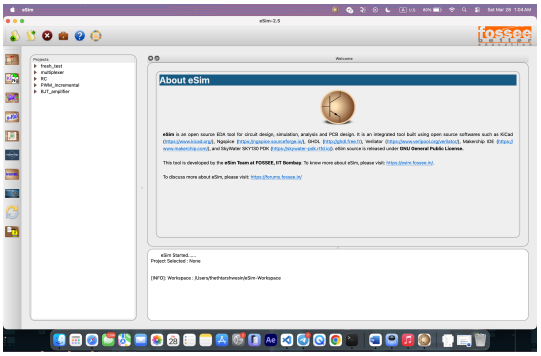

Figure 11.2: eSim home window

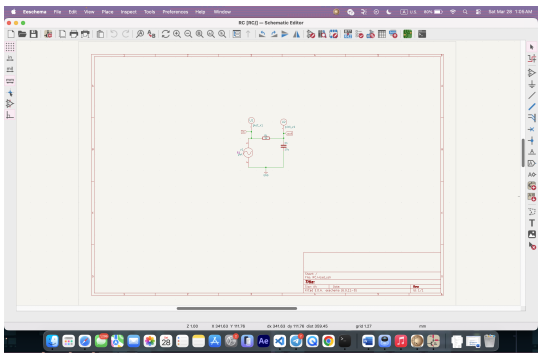

Figure 11.3: KiCad schematic call

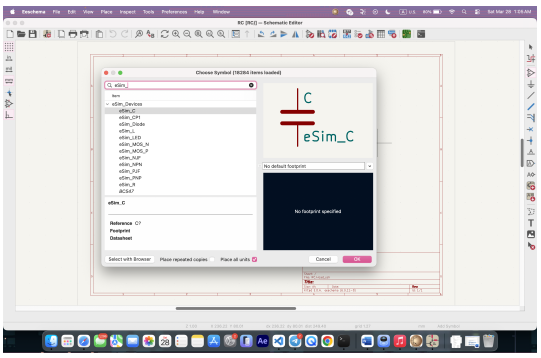

Figure 11.4: eSim symbols

11.3 ngspice Simulation Testing

A simple RC circuit was used to verify the full simulation pipeline within the `.app`.

1. A schematic was created using the embedded KiCad editor.
2. The netlist was exported and a transient analysis was run.
3. The simulation console was checked for output and the absence of code model errors.
4. The interactive ngspice plot window was verified to open via XQuartz.

Table 11.2: ngspice Simulation Test Results

Check	Method	Result
Simulation runs without crash	Console	Pass
No `.cm` loading errors	Console	Pass
`SPICE_SCRIPTS` correctly set	Console	Pass
Output waveform visible in console	Visual	Pass
Interactive plot window opens	Visual	Pass
`DISPLAY` resolved correctly	Console	Pass

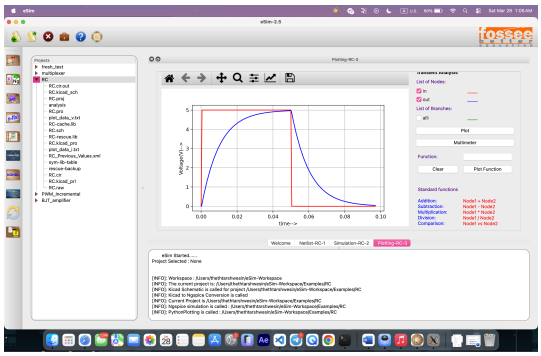

Figure 11.5: Python plotting

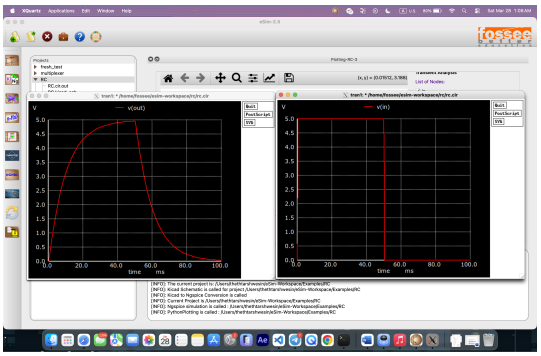

Figure 11.6: XQuartz ngspice plotting

11.4 KiCad Integration Testing

KiCad integration was verified by opening a schematic from within eSim and confirming that the `eeschema` editor launched correctly.

Table 11.3: KiCad Integration Test Results

Check	Method	Result
eeschema launches from eSim	Visual	Pass
eSim symbol libraries registered	KiCad UI	Pass
Schematic opens correctly	Visual	Pass
Netlist export succeeds	File	Pass

11.5 Installer Testing

The `eSim.pkg` installer was tested by running it on the development machine with the `.app` removed from `/Applications` first, simulating a fresh installation.

Table 11.4: Installer Test Results

Check	**Method**	**Result**
pkg opens with installer wizard	Visual	Pass
preinstall completes without error	Log	Pass
.app placed in /Applications	Finder	Pass
postinstall completes without error	Log	Pass
`~/.esim/` created	Terminal	Pass
`~/.nghdl/` created	Terminal	Pass
Quarantine attribute removed	Terminal	Pass
eSim launches after install	Visual	Pass

11.6 Known Limitations

The following limitations were identified during testing and are documented for completeness:

- **arm64 only** — The `.app` and `.pkg` were tested exclusively on Apple Silicon. Intel x86_64 compatibility has not been verified.
- **XQuartz required** — Interactive ngspice plot windows require XQuartz to be installed. If XQuartz is absent and `_ensure_display()` exhausts all fallbacks, plot windows do not open. Batch simulation and matplotlib-based plots are unaffected.
- **GHDL not bundled** — GHDL is not included in the `.app`. VHDL simulation features require GHDL to be separately installed.
- **Unsigned package** — The `.pkg` is not signed with an Apple Developer certificate. On machines with strict Gatekeeper settings, the user may need to explicitly allow the installer via System Preferences → Privacy & Security.

Chapter 12

Exploration of KiCad Plugins

12.1 KiCad 6 Plugin Architecture

KiCad 6 exposes a Python scripting interface through SWIG-generated bindings to its internal C++ data model, accessible as the `pcbnew` module inside KiCad's embedded Python interpreter. Plugins are discovered from platform-specific directories (e.g. `~/.config/kicad/6.0/scripting/plugins/` on Linux) and must subclass `pcbnew.ActionPlugin`, overriding `defaults()` and `Run()`, then call `register()` at module load time. Each registered plugin appears automatically under **Tools** → **External Plugins** in the PCB editor (pcbnew).

```
class ESimNetlistExporter(pcbnew.ActionPlugin):
    def defaults(self):
        self.name        = Export␣Netlist␣to␣eSim
        self.category    = eSim␣Integration
        self.description = Exports␣the␣board␣netlist␣as␣a␣SPICE␣.
           cir␣file.
        self.show_toolbar_button = True
    def Run(self):
        board = pcbnew.GetBoard()   # access active board
        # iterate footprints, reconstruct net topology, write .cir
           ...
```

12.2 Prototype Plugins Developed

Table 12.1: KiCad Plugin Integration Overview

Plugin	**Menu Entry**	**Function**
Netlist Exporter	Export Netlist to eSim	Board → SPICE `.cir` file
eSim Launcher	Launch eSim	Opens eSim with project directory
Simulation Trigger	Run eSim Simulation	Export + `ngspice -b` + results dialog

The **Netlist Exporter** iterates all footprints via `board.GetFootprints()`, maps pad numbers to net names using `pad.GetNetname()`, and writes standard Berkeley SPICE syntax. Net names are sanitised (hierarchical `/` prefix stripped; `GND`, `AGND`, `0V` → node `0`). The **Simulation Trigger** additionally invokes `ngspice` in batch mode (`-b` flag), captures `stdout`/`stderr`, and presents results in a scrollable `wx` dialog without leaving KiCad.

12.3 Key Findings and Limitations

The pcbnew API provides sufficient data for net-topology extraction but does *not* expose Eeschema's SPICE symbol fields (`Spice_Model`, `Spice_Node_Sequence`). A supplementary utility was developed to parse the richer KiCad XML intermediate netlist (exported from Eeschema), which does carry these fields. Additionally, KiCad 6 restricts Action Plugins to the PCB editor; adding eSim menu entries to the schematic editor requires the IPC API introduced in KiCad 7.

12.4 Conclusion and Next Steps

The prototype demonstrates that a functional KiCad → eSim workflow—netlist export, application launch, and batch simulation—is fully achievable within KiCad 6's plugin framework with modest Python code. Recommended next steps include: **(1)** enriching netlist export using Eeschema XML for model-accurate SPICE output; **(2)** adding a persistent settings dialog for eSim/ngspice paths; **(3)** migrating to the KiCad 7 IPC API to extend integration into the schematic editor; and **(4)** embedding a lightweight waveform viewer for ngspice `.raw` output directly inside KiCad.

Chapter 13

Remaining Work & Future Scope

13.1 GHDL Bundling

GHDL is the VHDL simulator invoked by eSim's nghdl module. It is currently not included in the `.app` bundle and must be separately installed by the user, which undermines the goal of a self-contained installer.

Bundling GHDL follows the same pattern as ngspice: build an arm64 binary from source, run `otool -L` to identify its shared library dependencies, copy those libraries into the bundle, relink with `@loader_path`, and ad-hoc codesign on Apple Silicon. A dedicated relinking script analogous to `relink_ngspice.sh` would be required.

13.2 Apple Code Signing and Notarization

The current `.pkg` is unsigned. For distribution outside the laboratory, two additional steps are required:

1. **Code signing** — The `.app` and all bundled binaries must be signed with an Apple Developer ID certificate using `codesign`. The current ad-hoc signing applied by `relink_ngspice.sh` satisfies Apple Silicon's requirement that modified binaries be signed, but does not constitute a trusted certificate signature.

2. **Notarization** — The signed `.pkg` must be submitted to Apple's notarization service using `notarytool`. Notarization attaches a ticket to the package that Gatekeeper verifies, allowing the installer to run on any Mac without security warnings and eliminating the need for the manual quarantine removal in `postinstall`.

13.3 Verilator Integration

Verilator is used by eSim for Verilog simulation. Like GHDL, it is currently invoked from the system `PATH` and is not bundled. Bundling Verilator presents similar challenges to GHDL — binary compilation, dylib resolution, and relinking — and is identified as a follow-on item.

Chapter 14

Conclusion

This project set out to accomplish something that had never been done before for eSim: make it installable on macOS. Prior to this work, a macOS user had no practical path to running eSim without manual compilation of every dependency and a technically involved environment setup. The project has addressed this gap end-to-end, from source code to a distributable `.pkg` installer.

The key technical contributions of this capstone are:

1. **ngspice 35 build for macOS** — A working build configuration for ngspice 35 on Apple Silicon, with X11 support, corrected `spinit` paths, and verified code model loading.

2. **macOS source code modifications** — Platform-specific fixes across six files in the eSim codebase, enabling correct tool invocation, path resolution, environment handling, and KiCad library registration on macOS.

3. **PyInstaller bundling** — A fully working `.app` bundle produced via PyInstaller, resolving the frozen-app `__file__` problem through the `esim_paths` module, and correctly bundling all assets, hidden imports, and external binaries.

4. **X11 dylib bundling and relinking** — All eleven XQuartz shared libraries identified, bundled, and relinked using `@loader_path`, making ngspice portable across machines regardless of XQuartz installation state.

5. **Runtime environment resolution** — Programmatic resolution of `DISPLAY`, `SPICE_SCRIPTS`, and ngspice binary paths, ensuring correct behaviour when eSim is launched from Finder rather than a terminal.

6. **.pkg installer** — A complete macOS installer built with `pkgbuild`, including `preinstall` dependency detection and `postinstall` configuration, quarantine removal, and permissions setup.

The result is a working `eSim.pkg` that installs eSim on Apple Silicon macOS with a single double-click, with no prerequisite developer tools required from the user. Simulation, schematic editing via KiCad, and ngspice interactive plot windows via XQuartz all function correctly from the installed bundle.

Bibliography

[1] FOSSEE Project: https://fossee.in

[2] eSim Project: https://esim.fossee.in

[3] GitHub Repository for eSim: https://github.com/FOSSEE/eSim

[4] GitHub Repository for NGHDL, the interface for enabling Mixed-Signal Simulations through eSim: https://github.com/FOSSEE/nghdl

[5] Ngspice Documentation: https://ngspice.sourceforge.io

[6] KiCad EDA Tool: https://kicad.org

[7] KiCad Previous Releases: https://downloads.kicad.org/kicad/macos/explore/stable

[8] Pre-built GHDL Download Page: https://github.com/ghdl/ghdl/releases: version 5.0.1 and onwards have ARM architecture support binaries

[9] Verilator Website: https://www.veripool.org/verilator/

[10] Pyinstaller Usage: https://pyinstaller.org/en/stable/usage.html