eSim Semester Long Internship
2025-2026

On

Develop a Tool and Package Manager for eSim

Submitted by:

Nu Htoo Wai

Bachelor of Engineering (Honours)
Electronics and Communications Engineering
Myanmar Institute of Information Technology, Myanmar

Under the guidance of:

Prof. Prabhu Ramachandran

Department of Aerospace Engineering
Indian Institute of Technology Bombay

26 March 2026

Acknowledgement

I express my sincere gratitude to **Prof. Prabhu Ramachandran** for providing me with the opportunity to participate in the FOSSEE internship programme and for his continued commitment to advancing open-source engineering education.

I also acknowledge **Prof. Kannan M. Moudgalya** for his foundational role in establishing the FOSSEE initiative and for creating the academic framework through which this internship was undertaken.

I am deeply grateful to my technical mentor, **Mr. Sumanto Kar**, for his consistent guidance, constructive feedback, and practical insights throughout the development of this project.

I extend my sincere appreciation to **Mrs. Usha Viswanathan** and **Mrs. Vineeta Parmar** for their unwavering support and encouragement throughout this internship. I am also thankful to **Mr. Varad Patil**, **Mrs. Shanthi Priya**, and the broader FOSSEE team for their technical insights and collaborative environment that made this work possible.

I extend my sincere gratitude to **Ms. Madhulika** and the entire **Spoken Tutorial team** for their guidance and support throughout the tutorial development process. Their meticulous review of the tutorial scripts was instrumental in ensuring the quality of the final deliverables.

I would like to thank my supervisor, **Prof. Dr Nu War**, for her guidance and support throughout my academic journey.I extend my sincere thanks to **Dr. Myat Thuzar Tun**, Pro-Rector, and **Dr. Win Aye**, Rector of Myanmar Institute of Information Technology, for their leadership and support.

Above all, I extend my deepest gratitude to my family for their unwavering belief in my abilities and constant encouragement. I am also profoundly grateful to **Sir Joe Htun Sein** for his mentorship and guidance, which have been instrumental in shaping my academic and professional path.

This internship provided me with significant hands-on experience in software development, GUI programming, cross-platform systems design, and technical content creation within a structured open-source environment.

Abstract

This report presents the development of a Tool and Package Manager for eSim, an open-source Electronic Design Automation (EDA) platform developed by FOSSEE, IIT Bombay. The project aims to simplify the installation, detection, updating, and removal of EDA tools required by the eSim platform across both Windows and Ubuntu operating systems.

The Tool Manager is built as a PyQt5 GUI application that provides a unified interface for managing tools including KiCad, Ngspice, LLVM, GHDL, Verilator, and eSim itself. The system supports real-time progress streaming, version selection, cross-platform detection, and seamless integration with the eSim application environment including toolbar-level launch support.

The project was developed in four progressive phases: Windows standalone, Ubuntu standalone, Windows eSim integration, and Ubuntu eSim integration with eSim main menu icon addition. Each phase built upon the previous, producing a cohesive and maintainable codebase.

In addition to the primary project, a supplementary contribution was made through the development of spoken tutorial scripts for the eSim tutorial series. A total of 40 tutorials were completed, covering the full range of eSim functionality from installation through to advanced mixed-signal simulation and OpenModelica integration.

Contents

Acknowledgement **1**

Abstract **2**

1 Introduction **7**

1.1 Background of the FOSSEE-eSim Environment 7
1.2 Problem Statement 8
1.3 Project Aim and Objectives 9
1.4 Project Scope and Development Phases 10
1.5 Technology Stack 11

2 System Architecture and Design **13**

2.1 Architectural Approach 13
2.2 System Architecture 13
2.2.1 GUI Layer 14
2.2.2 Subprocess Communication Model 15
2.2.3 Platform-Specific Backends 15
2.2.4 Tool Definition Structure 16
2.3 GUI Design 16
2.3.1 Real-Time Output Streaming 17
2.3.2 Threading Implementation 17
2.3.3 Double-Click Protection 18
2.4 State Management 18
2.5 Security Considerations 18

3 Windows Implementation **19**

3.1 Overview 19
3.2 Tool Detection on Windows 19
3.2.1 KiCad 19
3.2.2 Ngspice 20
3.2.3 LLVM 20
3.2.4 GHDL and Verilator 21

3.2.5 Chocolatey . . . 21
3.2.6 Detection Summary . . . 22
3.3 Tool Installation on Windows . . . 22
3.3.1 Chocolatey Auto-Installation . . . 22
3.3.2 KiCad . . . 22
3.3.3 Ngspice . . . 23
3.3.4 LLVM . . . 23
3.3.5 GHDL . . . 24
3.3.6 Verilator . . . 24
3.4 GUI Implementation . . . 25
3.5 Uninstall Window . . . 26
3.6 Admin Elevation . . . 27

4 Ubuntu Implementation 28
4.1 Overview . . . 28
4.2 Script Architecture . . . 28
4.3 Package Detection on Ubuntu . . . 29
4.4 Tool Installation on Ubuntu . . . 30
4.4.1 KiCad . . . 30
4.4.2 Ngspice . . . 30
4.4.3 GHDL . . . 31
4.4.4 Verilator . . . 31
4.4.5 Dependencies . . . 31
4.5 GUI Implementation . . . 32
4.6 Real-Time Progress Bar . . . 33
4.7 KiCad Preservation Fix . . . 34

5 eSim Integration 35
5.1 Windows eSim Integration . . . 35
5.1.1 eSim as a Managed Tool . . . 35
5.1.2 Launcher Interface: Four Tabs . . . 36
5.2 Ubuntu eSim Integration . . . 38
5.2.1 Toolbar Icon Addition . . . 38
5.2.2 Ubuntu Launcher Interface . . . 39
5.3 Post-Install Integration Steps . . . 40
5.4 Toolbar Integration Preparation for Windows . . . 40

6 Results and Testing 41
6.1 Testing Methodology . . . 41
6.2 Windows Test Results . . . 41

6.3 Ubuntu Test Results . . . 42
6.4 Issues Identified and Resolved During Testing . . . 42

7 Spoken Tutorial Development **44**
7.1 Overview . . . 44
7.2 Tutorial Format and Standards . . . 44
7.3 eSim 2.5 Updates . . . 44
7.4 Tutorials Completed . . . 45
7.5 Outputs . . . 47

8 Conclusion **48**
8.1 Summary of Work . . . 48
8.2 Lessons Learned . . . 49
8.3 Future Work . . . 49

List of Figures

1.1 The eSim 2.5 graphical interface on Ubuntu . . . 8

2.1 Subprocess-based architecture of the eSim Tool Manager . . . 14

3.1 Main Tool Manager window on Windows 10 . . . 25
3.2 Version selection dropdown active in the Windows GUI . . . 26
3.3 Uninstall window with per-tool checkboxes and status labels . . . 27
3.4 Uninstall window with confirmation dialog . . . 27

4.1 Main Tool Manager window on Ubuntu 22.04 . . . 32
4.2 Version selection in the Ubuntu Tool Manager . . . 33

5.1 Installation tab with Analog and Digital mode cards . . . 36
5.2 Management tab with launch button . . . 37
5.3 Uninstall tab with warning banner . . . 37
5.4 About tab displaying project information . . . 38
5.5 Comparison of eSim toolbar before and after Tool Manager integration . 39
5.6 Ubuntu Tool Manager Installation tab . . . 39
5.7 Package Updater showing version selection . . . 40

List of Tables

1.1 EDA tools within the scope of the eSim Tool Manager 10
1.2 Development and testing environment 12

2.1 Installation methods per tool per platform 16
2.2 Dashboard column structure 17

3.1 Windows tool detection methods and the problems they resolved 22
3.2 Windows tool installation methods 25

4.1 Ubuntu backend shell scripts and their roles 29
4.2 GHDL version to compiler dependency mapping 31
4.3 Keyword-to-progress mapping for the Ubuntu progress bar 33

6.1 Windows test results across all managed tools 41
6.2 Ubuntu test results across all managed tools 42
6.3 Issues identified and resolved during testing 43

7.1 Complete list of 40 spoken tutorials for eSim 2.5 47

Chapter 1

Introduction

1.1 Background of the FOSSEE-eSim Environment

The Free/Libre and Open Source Software for Education (FOSSEE) project is an initiative based at the Indian Institute of Technology Bombay (IIT Bombay), operating under the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education, Government of India. The mission of FOSSEE is to encourage the adoption of open-source software in engineering education and research by providing freely accessible, high-quality alternatives to proprietary platforms. Beyond software distribution, FOSSEE builds a community of student contributors through structured internship programmes and produces educational resources including spoken tutorials and textbook companions, ensuring that powerful engineering tools remain accessible to institutions regardless of budget or infrastructure constraints.

One of the most significant outcomes of the FOSSEE initiative is **eSim**, an open-source Electronic Design Automation (EDA) platform that supports circuit design, simulation, analysis, and PCB layout within a unified environment. eSim integrates six core open-source components into a single workflow: **KiCad** for schematic capture and PCB design, **Ngspice** for SPICE-based circuit simulation, **GHDL** for VHDL-based digital simulation, **Verilator** for Verilog and SystemVerilog simulation, **LLVM** for mixed-signal compilation via nghdl, and **Makerchip** for online Verilog development through the NgVeri interface. The platform has been widely adopted in academic institutions across India for laboratory instruction, independent learning, and research.

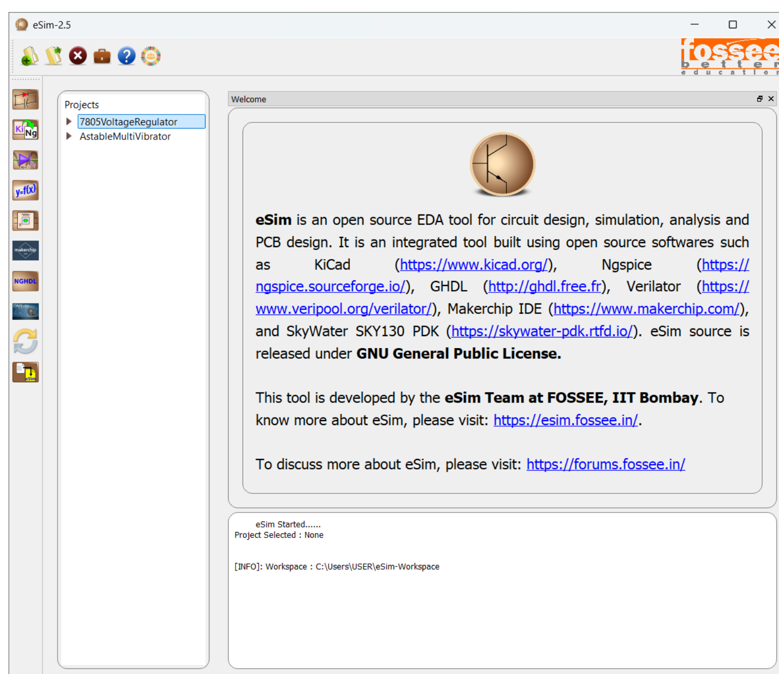

Figure 1.1: The eSim 2.5 graphical interface on Ubuntu

1.2 Problem Statement

Despite eSim's capabilities and wide adoption, a persistent and practical challenge exists that motivated this project: the installation and ongoing management of all the supporting tools that eSim depends on is a complex, error-prone, and time-consuming process that forms a genuine barrier to adoption.

Each of the six tools integrated into eSim must be installed separately, and every tool carries its own installer, version conventions, platform-specific behaviour, and dependency requirements. The process differs substantially between operating systems. On Windows, tools are typically handled through Chocolatey or direct binary downloads. On Ubuntu, tools are managed through the `apt` package manager, compiled source archives, or GitHub release binaries. In both cases, the process demands multiple manual steps, command-line familiarity, and administrative privileges.

Early investigation during the project revealed a consistent set of platform-specific problems that made even detection - let alone installation - more complex than anticipated. On Windows, calling `kicad -version` was found to open the KiCad graphical interface rather than returning a version string on some configurations. Calling `ngspice -v` was found to launch an interactive shell rather than printing version information. On Ubuntu, updating Ngspice through the apt package manager was found to silently uninstall KiCad as a side effect of dependency resolution. These were not theoretical concerns but concrete problems encountered during development that each required a dedicated solution.

Beyond the technical issues, users also have no unified view of which tools are installed, which versions are present, and what actions are needed. Each tool must be checked and managed independently, which is particularly problematic in classroom and laboratory environments where many machines need to be configured consistently. The time spent by students and instructors on installation issues directly reduces the time available for actual circuit design and learning.

1.3 Project Aim and Objectives

The aim of this project was to design, develop, and deploy a cross-platform **Tool and Package Manager for eSim** - a graphical application that provides a unified interface for managing all tools required by the eSim platform on both Windows and Ubuntu. The project was undertaken as a FOSSEE semester-long internship, with the work structured to produce a deliverable that could be integrated into the eSim ecosystem and made available to users.

The objectives set out at the beginning of the project were:

1. Design and implement a subprocess-based architecture that separates a shared GUI from platform-specific backend scripts
2. Develop reliable platform-appropriate detection methods for each tool, resolving the specific issues identified during investigation - including the KiCad GUI launch problem, the Ngspice interactive shell problem, and the LLVM PATH refresh issue on Windows.
3. Implement complete installation, update, and uninstall workflows for all six tools on both platforms, including handling of version-specific packages, offline use scenarios, and post-install verification.
4. Build a responsive PyQt5 GUI with real-time output streaming, threaded operations, and double-click protection, ensuring the interface remains usable during long-running installations.
5. Integrate the Tool Manager with the eSim application environment by developing a lightweight launcher suitable for the eSim toolbar and implementing the post-install steps required for full eSim compatibility.
6. Resolve the critical KiCad preservation bug on Ubuntu, where Ngspice updates were found to uninstall KiCad as an unintended side effect of apt dependency resolution.
7. Complete a supplementary contribution of 40 spoken tutorial scripts for the full eSim tutorial series, updated for eSim 2.5.

1.4 Project Scope and Development Phases

The project was structured into four progressive development phases, with each phase producing a testable and functional build before the next phase began.

Phase 1 Windows Standalone focused on building the core architecture from scratch: the PyQt5 GUI layout, the subprocess communication model, the pipe output format, the threading implementation, and the Windows backend covering detection and installation for all six tools. A significant portion of this phase was spent resolving the platform-specific detection issues described above, as these had to be solved before reliable detection could be achieved.

Phase 2 Ubuntu Standalone ported the system to Ubuntu 22.04 LTS, developing a completely separate Linux backend. This phase involved adapting detection to use `dpkg -l` queries, implementing APT-based installation workflows,adding auto-download with automatic compression format detection, and resolving the KiCad preservation bug.

Phase 3 Windows eSim Integration extended the Windows build to operate within the eSim application environment. This required adding eSim as a managed tool, implementing post-install integration steps, migrating all file paths to the standard eSim directory structure, and developing `main.py` as a lightweight launcher for the eSim toolbar.

Phase 4 Ubuntu eSim Integration with Icon Addition completed the Ubuntu build with equivalent eSim integration and added a Tool Manager launch button directly in the eSim main menu toolbar by modifying eSim's `Application.py` - the most complete form of the project delivered.

Tool	**Purpose in eSim**	**Windows**	**Ubuntu**
KiCad	Schematic capture and PCB design	Chocolatey	apt
Ngspice	SPICE circuit simulation	Chocolatey	apt + archive
LLVM	Mixed-signal compilation	Chocolatey	apt
GHDL	VHDL digital simulation	GitHub release	GitHub release
Verilator	Verilog simulation	.7z archive	pacman / apt
eSim	Main EDA application	FOSSEE server	FOSSEE server

Table 1.1: EDA tools within the scope of the eSim Tool Manager

macOS was considered during early planning. Homebrew provides a natural equivalent

to APT and Chocolatey on macOS, and the architecture was designed to accommodate a future macOS backend without changes to the GUI. However, the absence of macOS in the academic laboratory environments that eSim primarily targets, combined with the significant additional testing infrastructure required, led to macOS support being deferred. It is identified as a direction for future work.

1.5 Technology Stack

The Tool Manager is implemented entirely in Python, chosen for its cross-platform compatibility, strong subprocess management support, and suitability for PyQt5 GUI development. The key components of the technology stack and the reasoning behind each choice are as follows.

PyQt5 was selected as the GUI framework over alternatives such as Tkinter due to its more professional widget set, native-looking controls on both Windows and Ubuntu, and strong `QThread` support for threaded operations. The table widget, dropdown menus, and `QTextEdit` console area are all PyQt5 components.

subprocess with `Popen` was used for all backend invocations rather than `subprocess.run`, specifically to enable line-by-line stdout reading. This was a deliberate decision to support real-time output streaming to the GUI console - using `run` would have buffered all output until the process completed, making installations feel opaque and unresponsive.

QThread via the `CommandWorker` class handles all background operations. Running subprocess calls on the main GUI thread was found to freeze the interface during testing, which was addressed by moving all backend calls to `QThread` workers that communicate with the GUI exclusively through Qt signals.

winreg provided Windows registry access for KiCad detection, used when filesystem checks alone were insufficient due to non-standard installation paths.

py7zr handled extraction of `.7z` archives for the Verilator installation workflow on Windows, which relies on FOSSEE-hosted versioned archive packages.

Component	**Details**
Primary language	Python 3.10 and above
GUI framework	PyQt5
Windows test environment	Windows 10
Linux test environment	Ubuntu 22.04 LTS
Windows package manager	Chocolatey
Linux package manager	APT (Advanced Package Tool)
Archive handling	py7zr (.7z extraction)
Cross-platform testing	Windows Subsystem for Linux (WSL)
Target eSim version	eSim 2.5

Table 1.2: Development and testing environment

Chapter 2

System Architecture and Design

2.1 Architectural Approach

One of the first decisions made during this project was how to structure the relationship between the graphical interface and the platform-specific tool management logic. Two approaches were considered: a single codebase with platform-conditional branches throughout, or a shared GUI that delegates all tool operations to separate, self-contained backend scripts for each operating system.

The subprocess-based approach was chosen for several reasons. Because the backend runs as an independent child process, any crash or timeout in the backend is fully isolated from the GUI. The GUI can always report the outcome cleanly regardless of what happened internally. Each backend could also be developed, tested, and debugged entirely on its own platform without any dependency on the other. It also aligned naturally with how eSim itself organises platform support, using separate scripts for different operating systems.

This decision shaped every subsequent aspect of the project, from the communication protocol between GUI and backend to the threading model and state file design.

2.2 System Architecture

The architecture consists of four layers: a GUI layer, a platform-specific backend, package manager integrations, and a persistent state file. Figure 2.1 shows the relationship between these components.

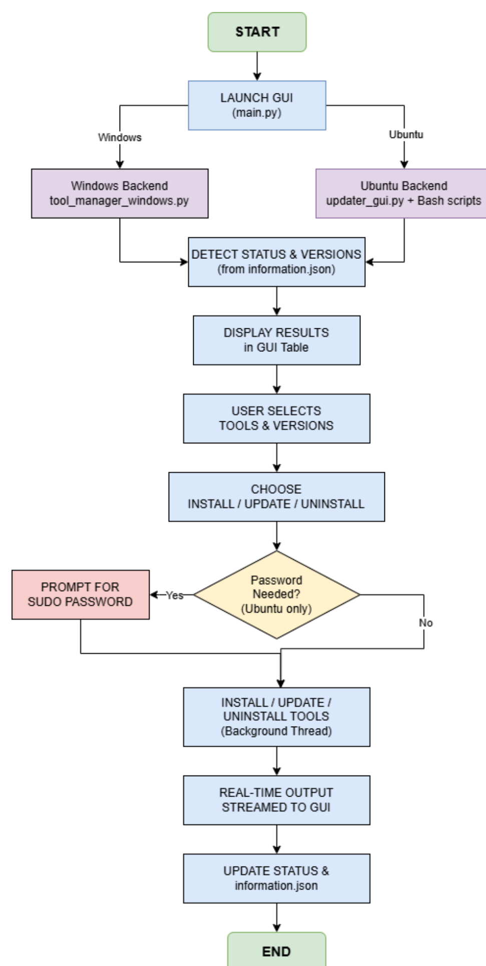

Figure 2.1: Subprocess-based architecture of the eSim Tool Manager

2.2.1 GUI Layer

The GUI layer differs between platforms. On Windows it is implemented in `gui.py` and communicates with a Python backend script via subprocess. On Ubuntu it is implemented in `updater_gui.py` and communicates with a set of Bash shell scripts via subprocess. Both GUIs follow the same design principles - table-based dashboard, `CommandWorker` threading, real-time output streaming, and pipe format status parsing - but the underlying communication targets differ. On Windows the subprocess call invokes a Python script with argparse commands. On Ubuntu the subprocess call invokes Bash scripts directly, each dedicated to a specific tool operation.

At startup, OS detection is performed using Python's `platform.system()` function and a `BACKEND` variable is set to the path of the appropriate backend. This is the only point at which the GUI distinguishes between platforms. All subsequent operations are platform-agnostic from the GUI's perspective.

2.2.2 Subprocess Communication Model

All backend operations are invoked through Python's `subprocess` module. When the user triggers an action, the GUI constructs a command and passes it to a `CommandWorker` thread, which launches the backend as a child process and reads its stdout line by line. Each output line is forwarded to the GUI console immediately.

The backend communicates its final result through a standardised output format printed at the end of each operation:

```
state | installed_version | target_version
```

The GUI scans backwards through the output to find the last line matching this pattern and uses it to update the dashboard row. The `state` field carries one of six values: `installed`, `not_installed`, `wrong_version`, `install_failed`, `update_failed`, or `error`.

Separating the streaming console output from the final status line was a deliberate decision. Early prototypes used the last line of stdout as the status, which broke whenever an install command produced a trailing blank line or warning message. The backwards scan for the pipe-delimited pattern made parsing robust regardless of trailing output.

2.2.3 Platform-Specific Backends

On Windows, `tool_manager_windows.py` is a self-contained Python script that accepts commands through `argparse` and handles all detection, installation, update, and uninstall logic using Chocolatey, GitHub releases, FOSSEE-hosted archives, and direct downloads.

On Ubuntu, the backend is a collection of dedicated Bash shell scripts, each responsible for a specific tool or operation. This script-based approach was chosen because the operations required on Ubuntu - building from source, managing PPAs, configuring LLVM versions, and running `make` - are naturally expressed as Bash workflows. Each script is also independently executable from the terminal, which was useful during development when individual tool installations needed to be debugged without launching the GUI.

Tool	**Windows**	**Ubuntu**
KiCad	Chocolatey (v7-9), direct GitHub download (v6)	APT via official KiCad PPA
Ngspice	Chocolatey	Built from source (SourceForge archive)
LLVM	Chocolatey	APT via official LLVM repository
GHDL	GitHub release ZIP download	Built from source (GitHub archive)
Verilator	FOSSEE-hosted `.7z` archive (specific versions), MSYS2 pacman (latest)	Built from source (GitHub archive)
eSim	FOSSEE server download (`static.fossee.in`)	Version-specific install script (`install-eSim-xx_xx.sh`)

Table 2.1: Installation methods per tool per platform

2.2.4 Tool Definition Structure

The set of managed tools and their supported versions is defined in a `TOOLS` dictionary at the top of `gui.py`. Each entry contains the tool name, a list of versions for the dropdown menu, and a description string. This centralised definition means that adding a new tool or a new supported version requires a change in exactly one place. The version lists are platform-aware where necessary - Verilator on Windows lists the specific FOSSEE-hosted archive versions, while on Ubuntu only the latest version is available through the package manager.

2.3 GUI Design

The dashboard presents a five-column table with one row per tool. The columns are described in Table **??**.

Column	Description
Tool with checkbox	Selecting the checkbox enables the version dropdown for that tool, preventing accidental version changes while browsing.
Version to install	Dropdown populated from the `TOOLS` dictionary. Defaults to `latest`.
Description	Brief description of the tool's role in eSim.
Installed version	Populated after a check operation. Shows the detected version, "Not installed", or "Installed (ver unknown)" if present but version cannot be parsed.
Status	Colour-coded indicator: green for installed, red for not installed or failed, orange for wrong version, blue for in-progress.

Table 2.2: Dashboard column structure

On startup, a `check` command is automatically dispatched for every tool, populating the installed version and status columns before the user takes any action. A "Refresh All" button re-triggers this check on demand.

2.3.1 Real-Time Output Streaming

An early version of the GUI used `subprocess.run` with `capture_output=True`, which collected all output internally and only made it available after the process completed. During testing this made the interface feel broken during long installations - the console showed nothing for several minutes, giving no indication of whether the process was running or had silently failed.

This was replaced with `subprocess.Popen` reading stdout line by line, with each line forwarded to the GUI console immediately. Users could now watch Chocolatey or apt output in real time, see which step was executing, and identify errors as they occurred.

2.3.2 Threading Implementation

Running subprocess calls on the main GUI thread froze the entire interface during any backend operation. This was addressed by implementing a `CommandWorker` class inheriting from PyQt5's `QThread`. Every backend call executes on a background thread and communicates back to the GUI through two Qt signals: a `progress` signal forwarding each output line to the console, and a `finished` signal carrying the complete output for status parsing.

2.3.3 Double-Click Protection

During testing, users would sometimes click an install button a second time before the first operation produced any visible output, triggering a second parallel installation of the same tool. The solution was an `active_tools` set that disables the Check, Install, and Update buttons while an operation is running for a given tool and re-enables them only when the `finished` signal fires.

2.4 State Management

A persistent state file, `information.json`, was introduced to solve a specific usability problem with the lightweight launcher. The launcher needed to show tool status instantly on startup - running subprocess detection calls for all tools at launch would take several seconds, which was unacceptable for a toolbar entry point. Reading from a local JSON file made startup essentially instant.

Each backend writes the installed version and installation date to `information.json` after every successful operation. The full GUI always runs fresh detection checks on startup and does not rely on the state file for its live dashboard. The state file is stored at `C:\FOSSEE\Tool-Manager\` on Windows and the equivalent path on Ubuntu, and its format is intentionally simple - a flat dictionary with one entry per tool containing a version string and an installation date.

2.5 Security Considerations

On Windows, administrator privileges are required for Chocolatey operations. The application checks for admin rights at startup using `ctypes.windll.shell32.IsUserAnAdmin()` and relaunches itself with a UAC prompt if not already elevated. Elevation is requested once at startup rather than per operation.

On Ubuntu, the sudo password is requested once per session through a masked `QInputDialog` and passed to backend subprocess calls via stdin. It is never written to disk or logged. All commands executed by the backend are fixed and predefined, eliminating the risk of command injection.

Chapter 3

Windows Implementation

3.1 Overview

The Windows standalone phase was the first completed phase of the project and the one that required the most investigative work. Before any installation logic could be written, reliable detection had to be achieved for each tool - and detection on Windows turned out to be significantly more complex than anticipated. Each tool presented its own platform-specific problem that had to be identified and solved before the backend could produce trustworthy results.

The Windows backend is implemented entirely in `tool_manager_windows.py`, a self-contained Python script that accepts commands through `argparse` and handles detection, installation, update, and uninstall for all six managed tools. The base directory for all Tool Manager files on Windows is `C:\FOSSEE\Tool-Manager\`, which was chosen to align with the directory structure used by the eSim ecosystem. A `Download\` subdirectory within this path serves as the local package cache for pre-downloaded installers and archives.

3.2 Tool Detection on Windows

3.2.1 KiCad

The first detection problem encountered during development was with KiCad. The natural approach of calling `kicad -version` to retrieve a version string was found to open the full KiCad graphical interface on some Windows configurations rather than printing to stdout and exiting. This blocked the subprocess call indefinitely and produced no usable output.

The solution developed was a two-step detection approach that avoids executing the KiCad binary entirely. The first step checks a set of known filesystem paths for the KiCad executable, covering the standard installation locations for versions 6 through 9:

- `C:\Program Files\KiCad\9.0\bin\kicad.exe`
- `C:\Program Files\KiCad\8.0\bin\kicad.exe`
- `C:\Program Files\KiCad\7.0\bin\kicad.exe`
- `C:\Program Files\KiCad\6.0\bin\kicad.exe`

If the filesystem check finds the executable, the version is inferred from the path rather than by running the binary. If the path-based check fails, a PATH lookup is attempted, and if the binary is found and can be invoked safely (i.e., does not launch the GUI), the version string is parsed from its output. This combination proved reliable across all tested Windows configurations.

3.2.2 Ngspice

Ngspice presented a similar problem with a different cause. Calling `ngspice -v` on Windows launches an interactive Ngspice shell rather than printing version information and exiting, blocking the subprocess call indefinitely. Executing the binary for detection purposes was therefore not viable.

The solution was to use Chocolatey's own query mechanism instead of the Ngspice binary. The command `choco list -exact ngspice` asks Chocolatey whether Ngspice is installed as a managed package and returns the version if found. Two command variants were implemented to handle compatibility between Chocolatey v1 and v2, since v2 removed the `-local-only` flag that was required in v1:

- `choco list -exact ngspice` (Chocolatey v2)
- `choco list -local-only -exact ngspice` (Chocolatey v1 fallback)

As a secondary fallback, a filesystem check for known Ngspice executable paths is performed. This catches Ngspice installations that were not managed through Chocolatey, though in those cases the version is reported as unknown since the binary cannot be safely invoked for version retrieval.

3.2.3 LLVM

LLVM detection encountered a different category of problem related to Windows PATH refresh behaviour. When LLVM is installed, its binaries are added to the system PATH. However, changes to the Windows PATH do not take effect in the current process session - they are only visible to processes launched after the environment has been refreshed. This means that if LLVM is installed during a Tool Manager session and detection is then

re-run in the same session, the binary would not be findable through PATH even though installation completed successfully.

To address this, LLVM detection was implemented to check known installation paths directly rather than relying on PATH resolution. The paths `C:\Program Files\LLVM\bin\clang.exe` and the equivalent `Program Files (x86)` path are checked first. If the binary is found at either location, it is invoked with `-version` to retrieve the version string. A `SendMessageTimeout` call to `HWND_BROADCAST` with `WM_SETTINGCHANGE` is also issued to trigger a partial environment refresh before the PATH search, which handles the subset of cases where the fixed paths check is insufficient.

3.2.4 GHDL and Verilator

GHDL and Verilator do not present the same detection issues as KiCad, Ngspice, and LLVM. Both tools respond correctly to version flag calls - `ghdl -version` and `verilator -version` - and exit cleanly. Detection for both tools uses a filesystem check for the known installation path followed by a version call if the executable is found. Since GHDL is installed to the Tool Manager's own `bin\` subdirectory, and Verilator is installed into the MSYS2 `mingw64` tree, the detection functions check these specific paths rather than relying on system PATH.

3.2.5 Chocolatey

Chocolatey itself is treated as a managed component of the system since it is a prerequisite for installing KiCad, Ngspice, and LLVM. Detection checks for the Chocolatey executable using `shutil.which` and, if found, invokes `choco -version` to retrieve the installed version. If Chocolatey is not found, the Tool Manager installs it automatically before proceeding with any Chocolatey-dependent operations.

3.2.6 Detection Summary

Tool	Detection method	Problem resolved
KiCad	Filesystem path check + PATH lookup	`kicad -version` opens GUI on some configurations
Ngspice	`choco list -exact ngspice` + filesystem fallback	`ngspice -v` opens interactive shell
LLVM	Fixed path check + `SendMessageTimeout` PATH refresh	PATH not updated mid-session after installation
GHDL	Filesystem check of Tool Manager bin dir + version call	Installed to non-standard path outside system PATH
Verilator	Filesystem check of MSYS2 mingw64 tree + version call	Installed inside MSYS2, not on system PATH
Chocolatey	`shutil.which` + `choco -version`	No binary execution issues; standard detection

Table 3.1: Windows tool detection methods and the problems they resolved

3.3 Tool Installation on Windows

3.3.1 Chocolatey Auto-Installation

Rather than requiring users to install Chocolatey manually before using the Tool Manager, a `install_chocolatey` function was implemented that handles this automatically. When a Chocolatey-dependent operation is requested and Chocolatey is not found, the function constructs a PowerShell command that downloads and executes the official Chocolatey installation script from `community.chocolatey.org`. The execution policy is set to bypass for the current process scope only, and the security protocol is set to TLS 1.2. After the PowerShell command completes, the installation is verified by re-running `find_chocolatey`.

3.3.2 KiCad

KiCad installation required significantly more work than the other tools due to a discovered issue with Chocolatey's download links for KiCad version 6. During testing, it

was found that Chocolatey's KiCad 6 package references download URLs from `osdn.net`, which has been taken offline. Attempting to install KiCad 6 through Chocolatey therefore always fails at the download step regardless of network conditions.

A separate direct download path was implemented for KiCad 6 that bypasses Chocolatey entirely. The function `install_kicad_direct` downloads the KiCad 6 installer directly from the official KiCad GitHub releases mirror, with a progress hook reporting download percentage and file size to the console in real time. The installer is then executed silently using NSIS silent install flags (`/S /NCRC`). Three version candidates are attempted in sequence (6.0.11, 6.0.10, 6.0.9) before reporting failure.

For KiCad versions 7, 8, and 9, Chocolatey is used with a three-step process. First, a thorough cleanup of any existing KiCad installation is performed - this involves a Chocolatey uninstall, a PowerShell script that terminates any running KiCad processes and removes all known KiCad installation directories, and a Chocolatey cache clear. A brief delay is introduced after cleanup to allow the filesystem to settle before the new installation begins. Second, the install command is run with version-specific candidates from the `KICAD_VERSIONS` mapping, trying each fallback version if the primary fails. Third, post-install verification checks both the detection function and a set of known installation paths to confirm success.

3.3.3 Ngspice

Ngspice installation begins with an uninstall of any existing Ngspice package to avoid version conflicts, then installs the requested version through Chocolatey using the `NGSPICE_VERSIONS` mapping to translate user-friendly version numbers to exact Chocolatey package versions. The installation is streamed in real time through `run_cmd_stream` and verified after completion using `find_ngspice_safe`.

3.3.4 LLVM

LLVM follows the same three-step pattern as KiCad 7-9: uninstall existing, install via Chocolatey with the exact version from `LLVM_VERSIONS`, then verify using fixed-path detection rather than PATH resolution. The fixed-path verification was specifically added to handle the PATH refresh problem described in the detection section - since PATH does not update mid-session, checking the known installation path directly is the only reliable way to confirm a successful installation within the same session.

3.3.5 GHDL

GHDL does not have a Chocolatey package, so installation downloads directly from GitHub releases. A verified URL mapping was built covering versions 4.0.0, 4.1.0, and 5.0.0 and above, with the archive format differing between versions - older releases use `ghdl-MINGW32.zip` while newer releases use `ghdl-mcode-version-mingw64.zip`. The zip archive is downloaded to a temporary directory with a progress hook, extracted, and the `ghdl.exe` binary is located within the extraction and copied to the Tool Manager's own `bin\` directory. Post-install verification runs `ghdl -version` to confirm the binary is functional.

3.3.6 Verilator

Verilator presented the most complex installation scenario on Windows. No official pre-built Windows binaries are available through standard package managers for specific older versions, and prior work at FOSSEE had established a pattern of providing versioned `.7z` archives for Verilator. The installation function `_install_verilator_from_7z` was implemented to mirror this approach.

The function checks the local `Download\` cache for the requested version's `.7z` archive. If found, it uses the `py7zr` library to extract the archive contents into a temporary directory, then copies the extracted files into the MSYS2 `mingw64` tree. This approach integrates Verilator into the MSYS2 environment that nghdl depends on, following exactly the structure established by the senior's implementation.

For the `latest` version, MSYS2's `pacman` package manager is used as the installation method, invoked through the `run_msys2_command` helper that executes commands inside the MSYS2 bash environment.

Tool	Install method	Notes
Chocolatey	PowerShell + official install script	Auto-installed if missing before any dependent operation
KiCad 6	Direct GitHub download + NSIS silent install	Implemented after Chocolatey's osdn.net URLs were found broken
KiCad 7-9	Chocolatey with version fallback candidates	Three-step: cleanup, install, verify
Ngspice	Chocolatey with version mapping	Uninstall-first approach to avoid version conflicts
LLVM	Chocolatey with version mapping	Fixed-path verification due to PATH refresh issue
GHDL	GitHub release ZIP download	Copies binary to Tool Manager bin directory
Verilator (specific)	py7zr extraction of FOSSEE .7z archive	Installs into MSYS2 mingw64 tree
Verilator (latest)	MSYS2 pacman	Requires MSYS2 installed at known path

Table 3.2: Windows tool installation methods

3.4 GUI Implementation

The main Tool Manager window was implemented in `gui.py` using a fixed size of 1150 by 560 pixels, chosen to accommodate the five-column table comfortably with a console output area below. Figure 3.1 shows the main window on Windows 10.

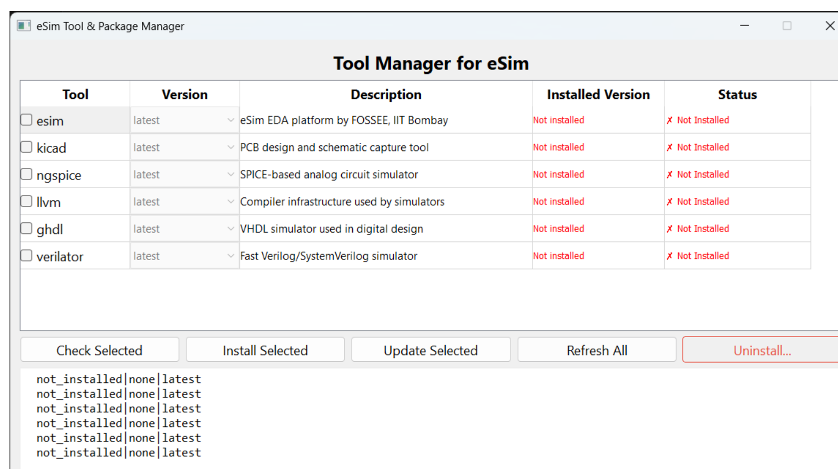

Figure 3.1: Main Tool Manager window on Windows 10

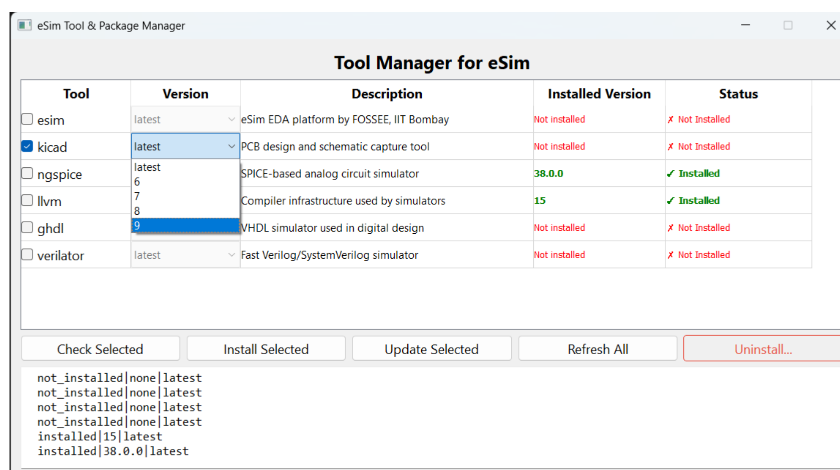

Figure 3.2: Version selection dropdown active in the Windows GUI

3.5 Uninstall Window

The uninstall functionality was implemented as a separate popup window rather than integrating it into the main dashboard. This was a deliberate decision made after considering that uninstalling a tool is a destructive operation that should require a deliberate user action rather than being reachable through the same controls used for installation.

The Uninstall window presents a list of all managed tools with a checkbox next to each one and a status label. A “Select All” button selects all tools at once. When the user clicks “Uninstall Selected”, a confirmation dialog appears listing the tools to be removed. Only after confirmation does the uninstall process begin, with live output streamed to a console within the Uninstall window.

Uninstall functions were implemented for all six tools using Chocolatey’s `uninstall` command for KiCad, Ngspice, and LLVM, the `delete` command for GHDL’s binary from the Tool Manager bin directory, a reversal of the MSYS2 extraction for Verilator, and eSim’s built-in `uninst-eSim.exe` uninstaller for eSim.

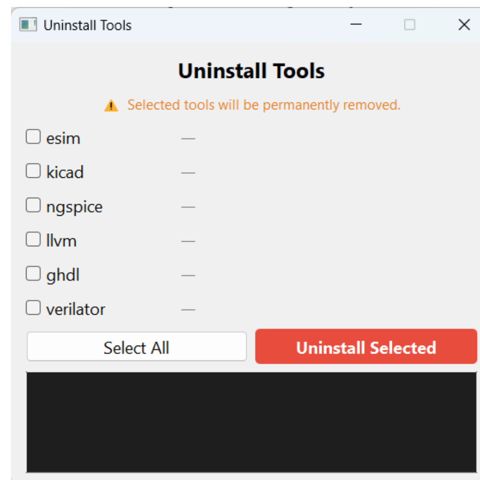

Figure 3.3: Uninstall window with per-tool checkboxes and status labels

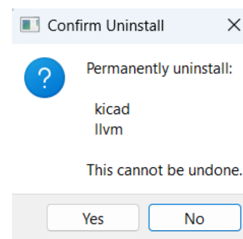

Figure 3.4: Uninstall window with confirmation dialog

3.6 Admin Elevation

All Chocolatey operations require administrator privileges. Rather than requesting elevation per operation - which would generate repeated UAC prompts - the application checks for admin rights at startup using `ctypes.windll.shell32.IsUserAnAdmin()`. If not already elevated, it relaunches itself using `ShellExecuteW` with the `runas` verb to trigger a single UAC prompt. All subsequent operations in the session run with the necessary privileges without further prompts.

Chapter 4

Ubuntu Implementation

4.1 Overview

The Ubuntu standalone phase ported the Tool Manager to Ubuntu 22.04 LTS, adapting the architecture to fit the Linux package management ecosystem. The most significant structural difference from the Windows implementation is that the Ubuntu backend is not a single Python script but a collection of dedicated Bash shell scripts, each responsible for a specific tool or operation. The GUI (`updater_gui.py`) invokes these scripts as subprocesses and streams their output to the console in real time, following the same subprocess communication model described in Chapter 2.

This script-based approach was chosen because the operations required on Ubuntu - building tools from source, managing PPAs, configuring LLVM versions, and running `make` - are naturally expressed as Bash workflows. Each script is also independently executable from the terminal, which proved useful during development and testing when individual tool installations needed to be debugged without launching the full GUI.

4.2 Script Architecture

The Ubuntu backend consists of the following shell scripts, each with a dedicated responsibility:

Script	Role
`check-packages-final.sh`	Checks installation status of all managed packages using `dpkg`, `which`, and version calls. Updates `information.json` with results.
`update-kicad-final.sh`	Installs or updates KiCad via PPA for the selected version. Handles PPA switching, cleanup, and custom eSim library copying.
`update-ngspice-final.sh`	Builds and installs Ngspice from source for versions 35 through 43. Creates a softlink at `/usr/bin/ngspice`.
`update-ghdl-dependency.sh`	Installs the correct LLVM, GNAT, and Clang versions for the selected GHDL version, then builds GHDL from source.
`update-verilator-final.sh`	Builds and installs Verilator from source for the selected version. Removes old installation before building.
`update-dependency-final.sh`	Installs and updates all Python and apt dependencies required by eSim including PyQt5, Matplotlib, Watchdog, Makerchip, and SandPiper.
`install-eSim.sh`	Top-level eSim installation script that detects the Ubuntu version and delegates to the appropriate version-specific installer.

Table 4.1: Ubuntu backend shell scripts and their roles

All scripts write their results back to `information.json` after each operation, recording the installed version and installation date. The GUI reads this file to update the dashboard after each operation completes.

4.3 Package Detection on Ubuntu

Detection on Ubuntu is handled by `check-packages-final.sh`, which checks each managed package using three methods in sequence. First, it queries `dpkg -l` to check whether the package is registered as installed in the apt database. Second, it uses `which` to check whether the package binary is available on the PATH. Third, it attempts to call the

binary with `-version` to confirm it is executable. If any of these checks succeeds, the package is marked as installed in `information.json`.

This three-layer approach was used because different tools are installed through different methods on Ubuntu. KiCad is installed via apt and is reliably detected by `dpkg`. Ngspice, GHDL, and Verilator are built from source and installed to `/usr/local/bin`, which is not always registered in the apt database, making `which` and version calls necessary as fallbacks.

4.4 Tool Installation on Ubuntu

4.4.1 KiCad

KiCad is installed via Ubuntu PPAs, with a dedicated PPA for each supported version. Three versions are supported: 6.0.11, 7.0.11, and 8.0.9. The installation process in `update-kicad-final.sh` follows these steps.

First, any existing KiCad installation is fully removed, including all related packages (`kicad-footprints`, `kicad-libraries`, `kicad-symbols`, `kicad-templates`), configuration directories, and all existing KiCad PPAs. This clean removal step was found to be necessary during testing, because leaving behind a previous version's PPA or configuration caused conflicts during subsequent installations.

Second, the appropriate PPA for the selected version is added and apt is updated. Third, KiCad is installed at the specific version string mapped to the user's selection. Fourth, post-install verification confirms the installation succeeded using `dpkg`. Finally, the custom eSim KiCad symbol library is deployed: the `sym-lib-table` is copied to the correct KiCad configuration directory (automatically detected from the installed version), old symbols are removed from `/usr/share/kicad/symbols/`, and the eSim-specific symbols are copied in their place.

4.4.2 Ngspice

Ngspice is built from source to support versions 35 through 43, a significant expansion from the three versions (38, 40, 43) supported in earlier iterations of the project. This range was determined by checking which versions were compatible with Ubuntu 22.04 and the nghdl framework.

The installation process in `update-ngspice-final.sh` begins by purging any existing Ngspice installation and installing the required build dependencies. The selected version's source archive is extracted from the local `nghdl/packages/` directory to a temporary directory, then the build process is run: `autoreconf` is called to regenerate the build

system if needed, followed by `configure`, `make`, and `make install`. After the build completes, a softlink is created at `/usr/bin/ngspice` pointing to the newly installed binary, and the PATH and profile files are updated to include the install directory.

4.4.3 GHDL

GHDL installation is the most complex operation in the Ubuntu backend because each GHDL version requires a specific combination of LLVM, GNAT, and Clang versions for compilation. This dependency matrix was established through testing and is encoded in `update-ghdl-with-dependency.sh`:

GHDL version	LLVM	GNAT	Clang
3.0.0	12	10	12
4.0.0	14	10	14
4.1.0	16	10	16
nightly	17	10	17

Table 4.2: GHDL version to compiler dependency mapping

Before installing GHDL, the script removes any previously installed Clang and LLVM packages, adds the official LLVM repository using the `llvm.sh` installer script, and installs the correct versions. LLVM environment variables (`PATH`, `LD_LIBRARY_PATH`, `C_INCLUDE_PATH`) are set before the build to ensure the compiler infrastructure is correctly located during the GHDL configure step.

GHDL itself is then built from the source archive using `./configure -with-llvm-config` pointing to the detected LLVM version, followed by `make` and `sudo make install`. The installed LLVM and GHDL versions are written back to `information.json` after each successful installation.

4.4.4 Verilator

Verilator is built from source for four supported versions: 4.228, 5.020, 5.026, and 5.030. The installation in `update-verilator-final.sh` begins by removing any existing Verilator binaries and directories from `/usr/local`. The required build dependencies (`make`, `autoconf`, `g++`, `flex`, `bison`, `ccache`) are installed, then the selected version's archive is extracted and the standard `configure`, `make`, `make install` sequence is run.

4.4.5 Dependencies

The `update-dependency-final.sh` script handles all Python and apt dependencies required by eSim that fall outside the main tool installations. These include

xterm, python3-psutil, python3-pyqt5, python3-matplotlib, python3-distutils, python3-pip from apt, and watchdog, hdlparse, makerchip-app, sandpiper-saas from pip.

4.5 GUI Implementation

The Ubuntu GUI is implemented in updater_gui.py and follows the same design as the Windows GUI: a table-based dashboard, a CommandWorker threading model, real-time output streaming to a console area, and pipe format status parsing. The key difference is that the subprocess calls launch Bash scripts rather than a Python backend.

Sudo privilege handling on Ubuntu requires a password for apt and build operations. The GUI prompts for the sudo password once per session through a masked QInputDialog, stores it in memory for the duration of the session, and passes it to backend subprocess calls via stdin. The password is never written to disk or logged.

Figure 4.1 shows the main Tool Manager window on Ubuntu 22.04, and Figure 4.2 shows version selection in use.

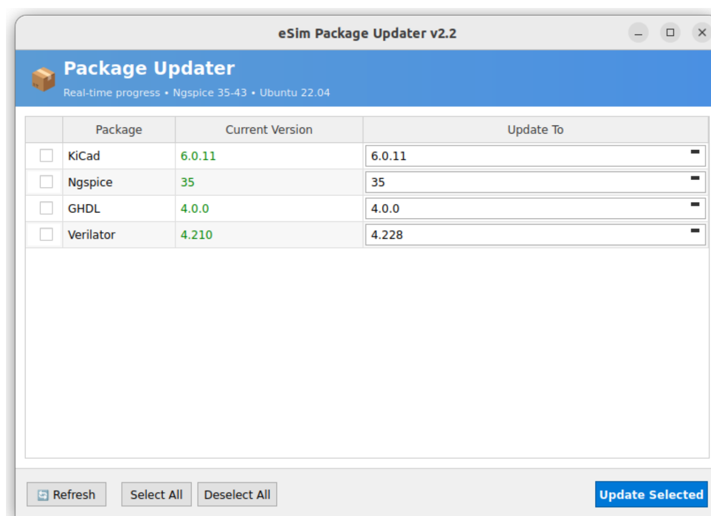

Figure 4.1: Main Tool Manager window on Ubuntu 22.04

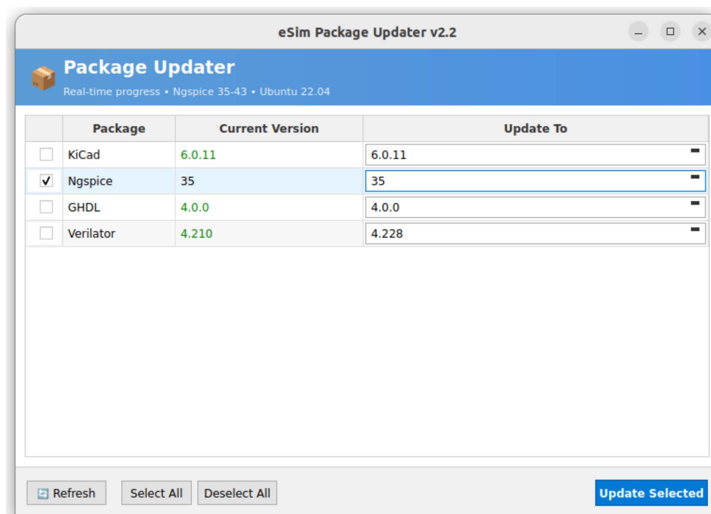

Figure 4.2: Version selection in the Ubuntu Tool Manager

4.6 Real-Time Progress Bar

One enhancement developed specifically during the Ubuntu integration phase was a real-time keyword-based progress bar that replaced the previous behaviour of the progress indicator jumping from 0% to 100% at the end of operations. This was particularly noticeable during long Ngspice and GHDL source builds, which could take several minutes and left users with no indication of progress.

The progress bar reads the stdout output of the running shell script line by line and maps specific output keywords to percentage milestones. The mapping used is shown in Table 4.3.

Table 4.3: Keyword-to-progress mapping for the Ubuntu progress bar

Output keyword	Progress value
Removing	10%
Installing dependencies	30%
Configuring	50%
Compiling	70%
Installing	85%
Success	100%

This approach does not require any changes to the shell scripts themselves - it works by parsing their natural output. The progress bar updates smoothly as each keyword is detected, giving users a clear and accurate sense of how far through a long build operation the system has progressed.

4.7 KiCad Preservation Fix

A critical bug was identified and resolved during Ubuntu integration testing. When the Ngspice update script removed `libngspice0`, apt's dependency resolution also removed `libngspice-kicad`, which triggered the uninstallation of KiCad. This happened silently - KiCad was present before the update but missing afterwards, with no obvious connection to the Ngspice operation.

The root cause was traced to the apt dependency chain between `libngspice-kicad` and KiCad. The fix modified the Ngspice update script to preserve `libngspice-kicad` while removing standalone `libngspice0` packages. A post-install verification check was also added to detect and automatically reinstall KiCad if affected.

This fix was one of the more significant debugging contributions of the project. The bug would have been very difficult for a user to diagnose independently since the connection between a Ngspice update and a missing KiCad is not at all obvious.

Chapter 5

eSim Integration

5.1 Windows eSim Integration

The Windows eSim integration phase extended the standalone Tool Manager to operate as a component within the eSim application environment. The standalone build produced in Phase 1 was fully functional as an independent application, but it had no connection to eSim itself - it could not manage eSim as a tool, did not follow eSim's directory conventions, and had no entry point accessible from within the eSim interface.

This phase addressed all three of those gaps. eSim was added as a managed tool with its own detection, installation, and uninstall functions. All file paths were migrated to match the standard eSim directory structure. A three-tier architecture was implemented with `main.py` serving as a lightweight launcher for quick status display, `gui.py` providing the full management interface, and `tool_manager_windows.py` handling all backend operations.

5.1.1 eSim as a Managed Tool

eSim itself was added as a managed tool in the GUI, with version options 2.2, 2.3, 2.4, and latest (2.5). Detection uses filesystem checks for known eSim installation indicators: the presence of `eSim.bat` and `uninst-eSim.exe`. The version is read from `information.json`, which is updated after successful installations.

Installation uses a hybrid download approach. The function first checks the local `Download` folder for the eSim installer corresponding to the selected version. If found locally, it is used directly. If not, it is downloaded from `static.fossee.in` with a live progress indicator. The installer is then executed silently with the `/S` flag.

Uninstallation invokes eSim's built-in `uninst-eSim.exe` uninstaller with the `/S` flag. If the uninstaller is not found, a PowerShell cleanup routine removes the eSim directory.

5.1.2 Launcher Interface: Four Tabs

The launcher (`main.py`) presents four tabs with a persistent status header showing installed versions with color-coded indicators.

The **Installation Tab** presents two visually distinct cards for Analog and Digital modes. Each card displays a description, feature list, estimated disk space, and an installation button. A collapsible progress panel appears during installation showing real-time console output.

Analog Mode installs eSim, KiCad (latest), and Ngspice (latest) for analog circuit simulation. **Digital Mode** adds GHDL, Verilator, and LLVM for digital simulation.

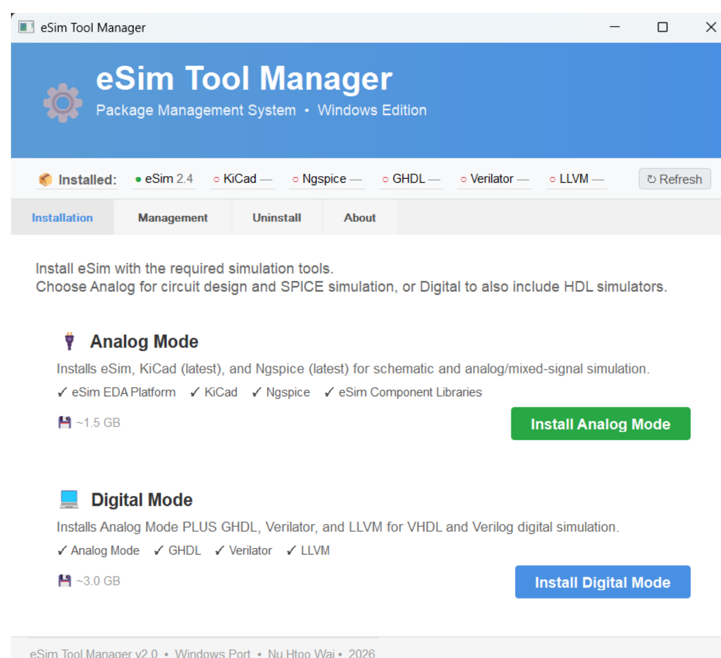

Figure 5.1: Installation tab with Analog and Digital mode cards

The **Management Tab** provides a button to open the full Tool Manager (`gui.py`) and an information panel listing version options.

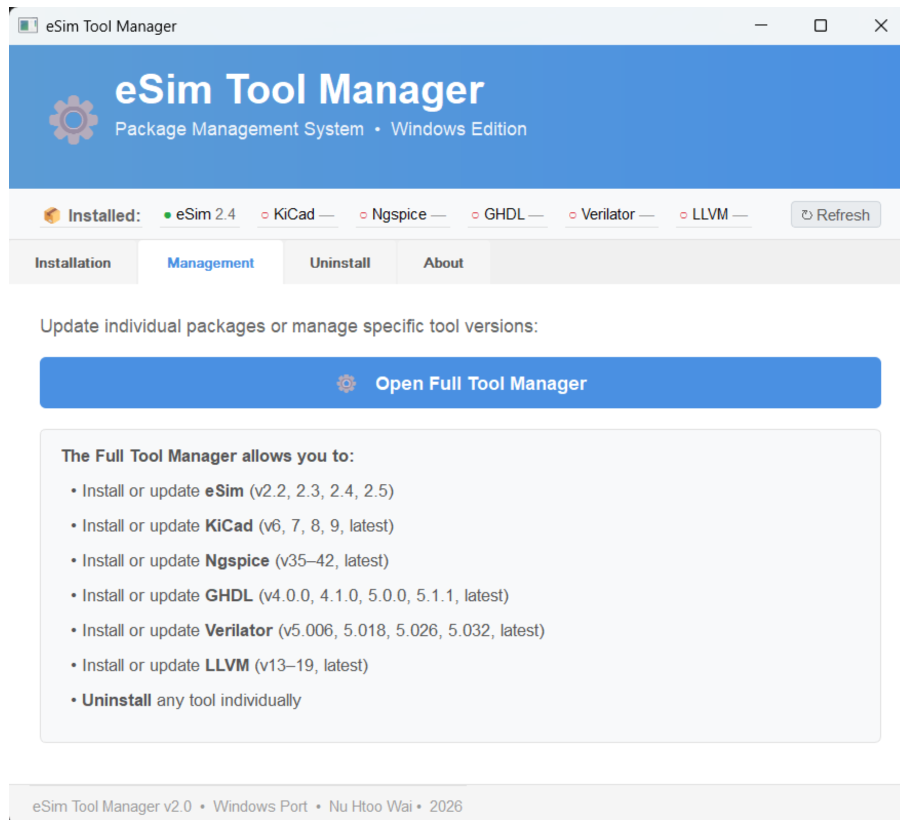

Figure 5.2: Management tab with launch button

The **Uninstall Tab** features a warning banner and three uninstall buttons: Uninstall Digital Packages (red), Uninstall Analog Packages (orange), and Uninstall Everything (gray). Each triggers a confirmation dialog before proceeding.

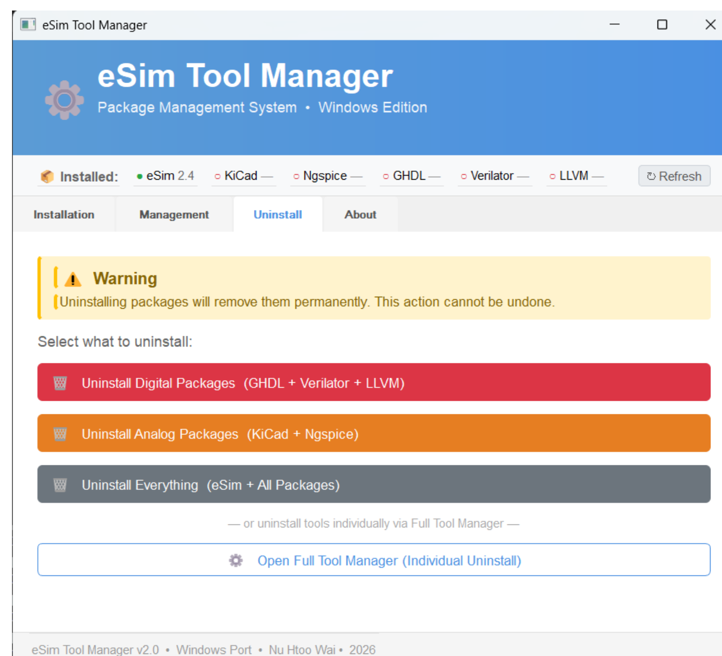

Figure 5.3: Uninstall tab with warning banner

The **About Tab** displays project information, key features, and supported package versions.

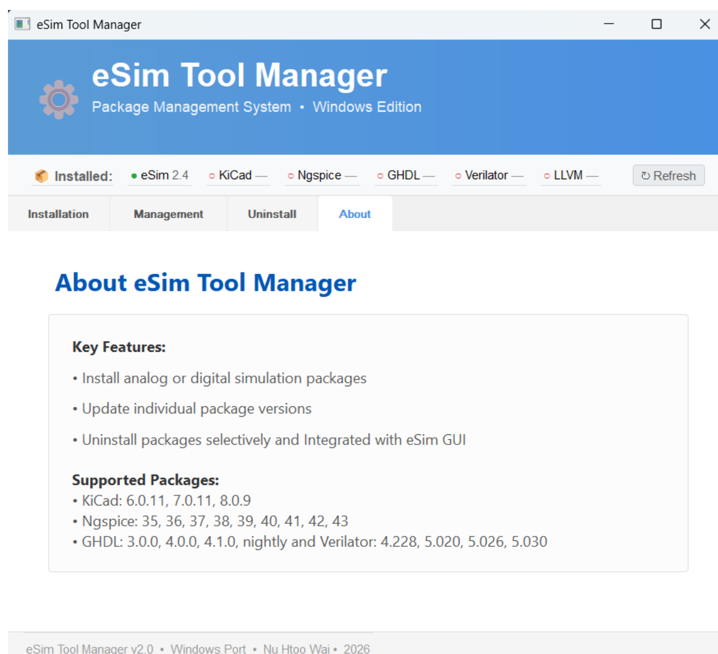

Figure 5.4: About tab displaying project information

5.2 Ubuntu eSim Integration

The Ubuntu eSim integration phase extended the Ubuntu standalone build to operate within the eSim application environment, with one significant additional deliverable: a Tool Manager launch button added directly to the eSim main menu toolbar. Unlike Windows, where the eSim source is compiled and inaccessible, the Ubuntu eSim distribution provides access to `Application.py`, the Python file that controls the eSim main window.

5.2.1 Toolbar Icon Addition

The most significant feature of the Ubuntu integration is the addition of a Tool Manager launch button directly in the eSim main menu toolbar. This was implemented by modifying `Application.py`, adding a new toolbar action that launches `main.py`.

Listing 5.1: Tool Manager button added to eSim's Application.py

```
1 # Tool Manager - Ubuntu 22.04 Enhancement
2 self.toolmanager = QtWidgets.QAction(
3     QtGui.QIcon(init_path + 'images/tool-manager.png'),
4     '<b>Tool␣Manager</b>', self
5 )
6 self.toolmanager.triggered.connect(self.open_toolmanager)
```

```

# Adding Action Widget to tool bar
self.lefttoolbar.addAction(self.toolmanager)
```

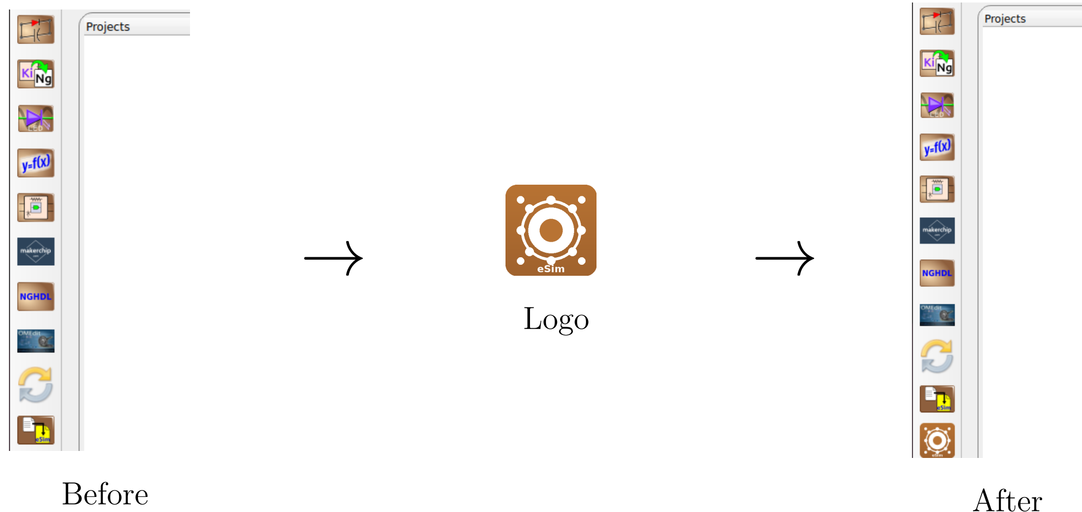

Figure 5.5: Comparison of eSim toolbar before and after Tool Manager integration

5.2.2 Ubuntu Launcher Interface

The Ubuntu launcher follows the same four-tab design as Windows, with the Management tab providing access to the Package Updater (`updater_gui.py`) for granular version control.

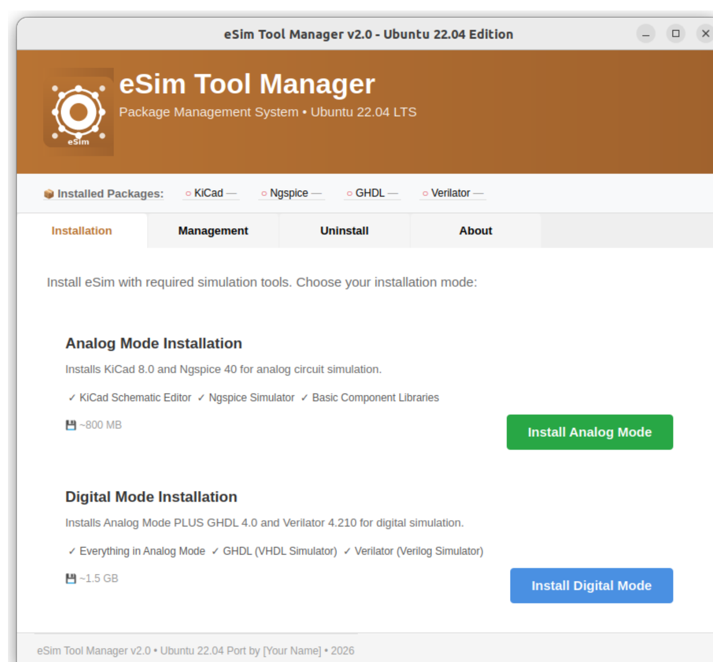

Figure 5.6: Ubuntu Tool Manager Installation tab

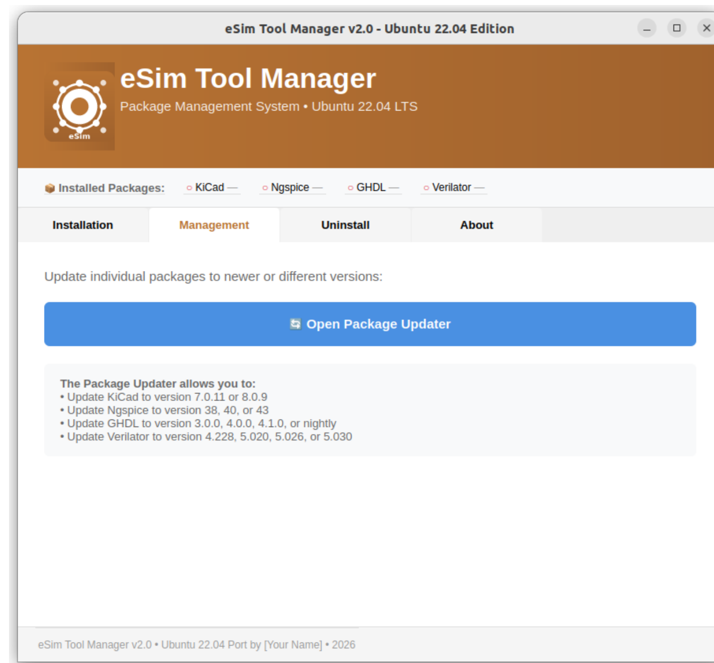

Figure 5.7: Package Updater showing version selection

5.3 Post-Install Integration Steps

After eSim installation completes, post-install integration steps are performed automatically on both platforms. These steps copy the custom KiCad symbol libraries and the `sym-lib-table` file to the correct eSim paths, ensuring the schematic editor can locate eSim-specific component symbols. The `libws2_32.a` library required for nghdl compatibility on Windows is also copied to its expected location.

5.4 Toolbar Integration Preparation for Windows

Since the Windows eSim distribution is compiled and does not expose `Application.py`, a file named `HOW_TO_ADD_TOOLBAR_BUTTON.txt` was created containing the exact code snippet to be inserted when source access becomes available.

Chapter 6

Results and Testing

6.1 Testing Methodology

Each phase of the project was tested on clean virtual machine environments to ensure that detection and installation functions performed correctly from a fresh system state with no prior tool installations. Testing on a clean environment was important because it is the most realistic representation of the situation a new eSim user would face.

The following test environments were used throughout the project: Windows 10 clean VM and Ubuntu 22.04 LTS clean VM, with no prior tool installations.

For each tool, the following sequence was tested in order: detection on a clean system (expected: not installed), installation of the selected version, detection after installation (expected: installed with correct version), update to a different version, and uninstall followed by a final detection check (expected: not installed).

6.2 Windows Test Results

All six tools were tested on Windows 10 across the full install-update-uninstall cycle. Table **??** summarises the outcomes.

Tool	**Detect**	**Install**	**Update**	**Uninstall**	**GUI stream**
KiCad	< 1 sec	10-15 min	10-15 min	< 1 min	✓
Ngspice	< 1 sec	2-3 min	2-3 min	< 1 min	✓
LLVM	< 1 sec	3-5 min	3-5 min	< 1 min	✓
GHDL	< 1 sec	2-3 min	2-3 min	< 1 min	✓
Verilator	< 1 sec	5-10 min	5-10 min	< 1 min	✓
eSim	< 1 sec	10-15 min	10-15 min	< 1 min	✓

Table 6.1: Windows test results across all managed tools

6.3 Ubuntu Test Results

All tools were tested on Ubuntu 22.04 LTS across the full cycle. Table ?? summarises the outcomes.

Tool	Detect	Install	Update	Uninstall	GUI stream
KiCad	$<$ 1 sec	5-8 min	5-8 min	$<$ 1 min	✓
Ngspice	$<$ 1 sec	10-15 min	10-15 min	$<$ 1 min	✓
GHDL	$<$ 1 sec	15-20 min	15-20 min	$<$ 1 min	✓
Verilator	$<$ 1 sec	8-12 min	8-12 min	$<$ 1 min	✓
eSim	$<$ 1 sec	3-5 min	3-5 min	$<$ 1 min	✓

Table 6.2: Ubuntu test results across all managed tools

6.4 Issues Identified and Resolved During Testing

Testing across both platforms revealed a number of issues that required investigation and fixes. The most significant are documented in Table 6.3.

Issue	Cause	Resolution
KiCad detection failing on Windows	`kicad -version` opens the GUI on some configurations	Replaced with filesystem path checks and registry lookups
Ngspice detection failing on Windows	`ngspice -v` opens an interactive shell	Replaced with `choco list -exact ngspice`
LLVM version check failing after install	Windows PATH does not refresh mid-session	Detection changed to read directly from known installation path
GUI freezing during installations	Subprocess calls running on main GUI thread	All backend calls moved to `QThread` CommandWorker
KiCad version 6 installation failing on Windows	Chocolatey's KiCad 6 package references broken download URLs	Implemented direct download fallback from GitHub releases

Issue	Cause	Resolution
KiCad uninstalled after Ngspice update on Ubuntu	Removing `libngspice0` triggered removal of `libngspice-kicad` via apt dependency resolution	Modified update script to preserve `libngspice-kicad`
Progress bar stuck at 0% on Ubuntu	Shell scripts called as single blocking processes	Switched to line-by-line output reading with keyword-based mapping
GHDL build failing on Ubuntu	GHDL versions require specific LLVM versions not installed by default	Implemented version-specific dependency matrix covering LLVM 12-17

Table 6.3: Issues identified and resolved during testing

Chapter 7

Spoken Tutorial Development

7.1 Overview

In addition to the primary Tool Manager project, a supplementary contribution was made through the development of spoken tutorial scripts for the eSim tutorial series. A total of 40 tutorial scripts were completed as part of this contribution, covering the full scope of eSim functionality from installation through to OpenModelica integration. All scripts were updated for eSim 2.5 on Ubuntu 22.04 and Windows.

7.2 Tutorial Format and Standards

Each spoken tutorial script follows a standardised two-table format used across the FOSSEE tutorial series. Table 1 contains the tutorial metadata including the tutorial title, duration, learning objectives, and prerequisites. Table 2 contains the Visual Cue and Narration columns, where each row describes what the narrator should show on screen and what they should say at that point.

The following standards were applied consistently across all 40 scripts: non-translatable terms formatted in bold; each sentence on a separate line for easy recording; an assignment added at the end of each tutorial; the EduPyramids outro replacing older slides; and a warm, beginner-friendly narration tone.

7.3 eSim 2.5 Updates

All scripts were verified against eSim 2.5 running on Ubuntu 22.04 and Windows. Several interface changes required updates: the "Append Schematic Sheet" menu option became "Insert Schematic Sheet Content"; netlist generation moved to **File** → **Export** → **Netlist**; the Docker installation method changed to use `run_esim_docker.py`.

7.4 Tutorials Completed

A total of 40 spoken tutorial scripts were completed for eSim 2.5, as listed below.

No.	Tutorial Title	Key Sub-Topics
1	eSim Linux Installation	Download, install, launch eSim
2	eSim Windows Installation	Download .exe, install, launch eSim
3	eSim Docker Installation	Download image, install using Docker, launch
4	Test run an example	Half wave rectifier schematic, simulation, Ngspice and Python plots
5	Schematic Creation with eSim	New project, add components, connect wires, annotate, ERC
6	Netlist Generation and Simulation	Generate netlist, add analysis parameters, convert to Ngspice, simulate
7	Semiconductor and Subcircuit Simulation	Add device models, subcircuit files, convert and simulate
8	Astable Multivibrator: Schematic	Create circuit, correct wire mistakes, junctions, plot components
9	Astable Multivibrator: Simulation	Transient parameters, convert netlist, assign device models, Python plotting
10	PCB Prep: Schematic Conversion	Remove sources/labels, add connectors, annotation, ERC
11	Footprint Mapping and Association	Launch Cvpcb, assign footprints, generate .net file
12	PCB Layout: Placement and Routing	Launch Pcbnew, move/orient footprints, board outline
13	Design Rule Check and Drills	Set DRC rules, track width, verify drill size and shape
14	Advanced Routing and Ground Planes	Place tracks, add ground plane outline and fill, perform DRC
15	Dimensions and Gerber Generation	Place dimensions, add text, generate and view Gerber files

Continued on next page

No.	Tutorial Title	Key Sub-Topics
16	Astable PCB: Layout	Launch Eeschema and Pcbnew, place tracks
17	Astable PCB: Planes and DRC	Add board outline, ground plane, perform DRC
18	Astable PCB: Gerber Output	Generate and view Gerber files
19	Creating Device Models	Introduction, create Germanium Diode 1N34A model
20	Simulating Device Models	Open Diode Characteristics example, convert, verify characteristics
21	Editing Device Models	Edit emission coefficient N, observe knee voltage changes
22	Uploading External Libraries	Download Schottky library, upload, save to User Libraries
23	Simulating External Libraries	Add library file, generate Ngspice and Python plots
24	Subcircuit Design and Netlisting	Create Half Adder subcircuit, connect ports, generate netlists
25	Symbol Creation and Library Editor	Create part library/symbol, add pins
26	Testing The Created Component	Verify in KiCad library, simulate using HA_Test
27	Modifying Subcircuit File	Edit Half Adder, add inverter gate, regenerate netlists
28	Updating Symbols in Library	Edit in Library Editor, save to eSim_subckt libraries
29	Simulating Edited Subcircuit	Verify modified subcircuit with Half Adder example
30	Uploading Spice Subcircuit File	Upload FA using NAND.sub, verify, simulate Fulladder
31	Introduction to UJT Subcircuit	.include statement, RMOD and Emitter model definition
32	Mixed Signal: NGHDL Part 1	Upload VHDL file, add/remove dependency files

Continued on next page

No.	Tutorial Title	Key Sub-Topics
33	Mixed Signal: NGHDL Part 2	Open full_adder, interface analog/digital components, simulate
34	Mixed Signal: NgVeri Part 1	Makerchip tab options, run Verilog to NgSpice converter
35	Mixed Signal: NgVeri Part 2	Create schematic, analog/digital blocks, KiCad to Ngspice
36	Makerchip IDE	Edit in Makerchip, compile, run, view waveforms, add signals
37	Advanced NgVeri: Dependencies	Create full_adder and 8bit multiplier models, Add File/Folder
38	Advanced NgVeri: Linting Flags	Edit lint_off, EOFNEWLINE, Edit modlst, remove models
39	OpenModelica: Modelica Converter	Launch converter, open project, convert netlist to model file
40	OpenModelica: Simulation	Simulate model, add and analyze plots

Table 7.1: Complete list of 40 spoken tutorials for eSim 2.5

.

7.5 Outputs

For each tutorial, the following deliverables were produced: tutorial scripts in the standard two-table format, assignment files supporting the assignment section, video recordings demonstrating each step, and reference files such as `esim_docker_commands.txt` and `Diode_1N4148.lib`.

Chapter 8

Conclusion

8.1 Summary of Work

This project delivered a fully functional, cross-platform Tool and Package Manager for the eSim EDA platform. The system was developed across four progressive phases, and the final result is an application that allows users to detect, install, update, and uninstall all tools required by eSim on both Windows and Ubuntu from a single graphical interface.

The Windows implementation handles tool management through a Python backend using Chocolatey for KiCad, Ngspice, and LLVM, GitHub releases for GHDL, FOSSEE-hosted archives for Verilator, and a hybrid download approach for eSim itself. Key contributions on Windows include resolving the KiCad GUI launch problem, the Ngspice interactive shell problem, the LLVM PATH refresh issue, and implementing a direct download fallback for KiCad version 6.

The Ubuntu implementation handles tool management through a set of dedicated Bash shell scripts, building most tools from source to support specific versioned installations. Key contributions on Ubuntu include the expansion of Ngspice version support from 3 to 9 versions, the GHDL version-to-compiler dependency matrix, and the resolution of the KiCad preservation bug.

Both builds were integrated with the eSim application environment. A lightweight launcher was developed as the eSim toolbar entry point. On Ubuntu, a Tool Manager launch button was added directly to the eSim main menu toolbar by modifying `Application.py`. On Windows, the integration was prepared for implementation when source access becomes available.

As a supplementary contribution, 40 spoken tutorial scripts were completed for the full eSim tutorial series, all updated for eSim 2.5.

8.2 Lessons Learned

Several important lessons emerged from the development work across this project. Cross-platform development always reveals edge cases that cannot be anticipated from documentation alone - the KiCad detection problem, the Ngspice shell behaviour, and the KiCad preservation bug were all discovered only through testing on real machines.

The architectural decision to use separate backend scripts per platform rather than a single conditional codebase proved to be the right call. It kept each backend clean and focused, made platform- specific debugging straightforward, and meant that a bug on one platform had no effect on the other.

Threading and real-time output streaming have a disproportionately large impact on perceived application quality. The difference between a GUI that freezes silently during an installation and one that streams live output is significant from a user experience perspective.

8.3 Future Work

Several directions were identified for future development:

- **Windows toolbar integration:** Completing the toolbar button integration when eSim Windows source access becomes available
- **macOS support:** Adding a macOS backend using Homebrew, completing cross-platform coverage
- **Automatic update checking:** Implementing background checking for new tool versions with user notifications
- **Expanded version support:** Extending version lists as new tool releases become available
- **Packaging and distribution:** Creating standalone installers or bundling with future eSim releases
- **Remaining spoken tutorials:** Extending the tutorial series to cover new eSim features as they are added

Bibliography

[1] FOSSEE Project, IIT Bombay. *Free/Libre and Open Source Software for Education.* Available: https://fossee.in

[2] eSim - Open Source EDA Tool for Circuit Design and Simulation. Available: https://esim.fossee.in

[3] KiCad EDA - Schematic Capture and PCB Layout. Available: https://kicad.org

[4] Ngspice - Open Source Spice Simulator. Available: https://ngspice.sourceforge.io

[5] GHDL - Open Source VHDL Simulator. Available: https://ghdl.github.io/ghdl

[6] Verilator - Fast Verilog/SystemVerilog Simulator. Available: https://verilator.org

[7] LLVM Compiler Infrastructure. Available: https://llvm.org

[8] Chocolatey - The Package Manager for Windows. Available: https://chocolatey.org

[9] APT (Advanced Package Tool) - Debian Package Manager. Available: https://wiki.debian.org/Apt

[10] PyQt5 - Python Bindings for Qt5. Available: https://riverbankcomputing.com/software/pyqt

[11] Python Software Foundation. Available: https://www.python.org

[12] MSYS2 - Software Distribution and Building Platform for Windows. Available: https://www.msys2.org

[13] FOSSEE Internship Reports and Documentation. Available: https://fossee.in/internship-reports

[14] Spoken Tutorial Project, IIT Bombay. Available: https://spoken-tutorial.org

[15] National Mission on Education through ICT (NMEICT), Ministry of Education, Government of India. Available: https://www.education.gov.in/nmeict