Summer Fellowship Report

On

User Interface Improvements and Debugging in eSim

Submitted by

Anup Kumar Pandey

Under the guidance of

Sumanto kar

IIT Bombay

July 14, 2025

Acknowledgment

We take this opportunity to express our sincere gratitude to FOSSEE, IIT Bombay, for providing us with the platform to contribute to the development of eSim 2.4. This internship offered us a unique opportunity to work on graphical user interface (GUI) improvements and bug fixing, deepening our understanding of software development practices within the domain of open-source EDA tools. Through this experience, we gained valuable insights into the importance of user-centric design, usability enhancements, and software stability in circuit simulation tools. The hands-on work on GUI refinement not only enhanced our technical proficiency but also sharpened our debugging and interface design skills—crucial competencies in modern electronic design environments.

We extend our heartfelt appreciation to Prof. Kannan M. Moudgalya for his vision and continued support of open-source innovation. Being part of this initiative has been both inspiring and intellectually rewarding.

A special note of thanks to our mentor, Mr. Sumanto Kar, for his constant guidance, timely feedback, and technical expertise. His support at every stage enabled us to tackle challenges effectively and complete our tasks with clarity and confidence. His mentorship played a vital role in shaping our approach to problem-solving and collaborative development.

This internship has been a highly enriching journey, preparing us for future endeavors in the field of electronics and software systems. We are immensely grateful for this opportunity and look forward to applying the knowledge gained in impactful ways in our careers ahead.

Contents

1 INTRODUCTION **5**

1.1 FOSSEE: Promoting Open-Source Software for Education 5

1.1.1 Overview of FOSSEE Initiative 5

1.1.2 Mission and Vision 5

1.1.3 Impact on Educational Technology 6

1.2 eSim: An Open-Source EDA Tool 7

1.2.1 Introduction to eSim 7

1.2.2 Key Features of eSim 7

1.2.3 Components Integrated with eSim 9

1.3 Project Scope and Objectives 9

1.3.1 Enhancement Goals 9

1.3.2 Target Improvements 10

1.3.3 Expected Outcomes 10

2 LITERATURE SURVEY **11**

2.1 NgSpice 11

2.1.1 Overview and Architecture 11

2.1.2 Simulation Capabilities 11

2.1.3 Integration with eSim 11

2.1.4 Performance Analysis 12

2.2 NGHDL (NgSpice + GHDL) 12

2.2.1 Mixed-Signal Simulation 12

2.2.2 Digital-Analog Interface 12

2.2.3 VHDL Integration 13

2.2.4 Current Working Status 13

2.3 User Interface Improvements and Debugging in eSim 13

2.3.1 Current UI Limitations 13

2.3.2 Theme Management Systems 13

2.3.3 Debugging Methodologies 14

2.3.4 Code Quality Enhancement 14

2.4 Related Work and Technologies 14

2.4.1 Open-Source EDA Tools Comparison 14

2.4.2 GUI Framework Analysis 15

2.4.3 Theme Implementation Strategies 15

3 PROBLEM STATEMENT **16**
3.1 Problem Statement . . . 16
3.1.1 Project Name Validation Issue . . . 16
3.1.2 Missing Delete Functionality . . . 17
3.1.3 User Interface Limitations . . . 17
3.2 Approach . . . 18
3.2.1 Problem Analysis Methodology . . . 18
3.2.2 Solution Design Strategy . . . 18
3.2.3 Implementation Framework . . . 18
3.2.4 Testing and Validation Approach . . . 18

4 IMPLEMENTATION **19**
4.1 Phase 1: Namespace Error Resolution . . . 19
4.1.1 Analysis of validation.py . . . 19
4.1.2 Analysis of newProject.py . . . 20
4.1.3 System Consistency and Testing . . . 20
4.2 Phase 2: Delete Functionality Implementation . . . 21
4.2.1 Analysis of User Requirements . . . 21
4.2.2 UI and Logic Implementation . . . 22
4.2.3 System Consistency and Testing . . . 23
4.3 Phase 3: GUI Theme Enhancement (Application.py as the Central Controller) . . . 23
4.3.1 Theme System Architecture . . . 23
4.3.2 Light Theme Implementation . . . 24
4.3.3 Dark Theme Implementation . . . 25
4.4 Phase 4: Component-Specific Enhancements . . . 26
4.4.1 Makerchip Integration . . . 26
4.4.2 NgVeri Enhancements . . . 29
4.4.3 Modelica Converter . . . 32
4.4.4 KiCad to NgSpice Converter . . . 35
4.4.5 Model Editor . . . 38
4.4.6 Schematic Converter . . . 41

5 RESULTS AND TESTING **45**
5.1 GUI Theme Implementation . . . 45
5.1.1 Light and Dark Mode Integration . . . 45
5.1.2 Color Scheme and Styling . . . 45
5.1.3 Component Coverage . . . 47
5.2 Additional Feature Testing . . . 47
5.2.1 Project Name Validation . . . 47
5.2.2 Delete Functionality Validation . . . 48
5.3 Performance Evaluation . . . 48
5.3.1 Responsiveness Analysis . . . 48
5.3.2 Memory and Resource Usage . . . 48
5.3.3 Application Load Time . . . 49

6 CONCLUSION AND FUTURE WORK **50**
6.1 Conclusion 50
6.1.1 Project Summary 50
6.1.2 Achievements 50
6.1.3 Challenges Overcome 50
6.2 Future Scope 51
6.2.1 Planned Enhancements 51
6.2.2 Technology Upgrades 51
6.2.3 Community Contributions 51
6.3 Recommendations 51
6.3.1 Development Best Practices 51
6.3.2 Maintenance Guidelines 52
6.3.3 Extension Possibilities 52

Chapter 1

INTRODUCTION

1.1 FOSSEE: Promoting Open-Source Software for Education

1.1.1 Overview of FOSSEE Initiative

FOSSEE (Free/Libre and Open Source Software for Education) project promotes the use of FLOSS tools in academia and research. The FOSSEE project is part of the National Mission on Education through Information and Communication Technology (ICT), Ministry of Education (MoE), Government of India. This initiative, based at the Indian Institute of Technology Bombay (IIT Bombay), represents a significant step towards reducing dependency on proprietary software in educational institutions across the country.

The FOSSEE project was conceived to address the growing need for accessible, high-quality educational software that institutions could freely use, modify, and distribute. By focusing on Free/Libre and Open Source Software (FLOSS), the initiative aims to democratize access to advanced technological tools that were previously available only through expensive proprietary licenses. The project encompasses various domains including scientific computing, programming languages, electronic design automation, and computer-aided design tools.

The initiative encourages the use of FLOSS tools through various activities including workshops, training programs, textbook companions, and software development projects. These activities are designed to build capacity among educators and students, enabling them to effectively utilize open-source alternatives to commercial software packages.

1.1.2 Mission and Vision

The primary mission of FOSSEE is to improve the quality of education in India by promoting the adoption of Free/Libre and Open Source Software. The project aims to reduce dependency on proprietary software in educational institutions. This mission is driven by several key factors:

Accessibility and Affordability: By promoting FLOSS tools, FOSSEE makes advanced software accessible to institutions that may not have the financial resources

to purchase expensive proprietary licenses. This democratization of technology ensures that quality education is not limited by economic constraints.

Customization and Flexibility: Open-source software provides the flexibility to modify and adapt tools according to specific educational needs. This allows institutions to customize software to align with their curriculum requirements and teaching methodologies.

Knowledge Transfer and Skill Development: By working with open-source tools, students and educators gain deeper insights into software architecture and implementation. This knowledge transfer is crucial for developing technical expertise and fostering innovation.

Long-term Sustainability: Unlike proprietary software that may become obsolete or unavailable, open-source tools ensure long-term sustainability of educational resources. The community-driven development model ensures continuous improvement and support.

The vision of FOSSEE extends beyond mere software adoption to creating a comprehensive ecosystem where Indian educational institutions can leverage cutting-edge technology for research and development. This vision includes building local expertise in software development, encouraging contributions to global open-source projects, and positioning India as a leader in educational technology innovation.

1.1.3 Impact on Educational Technology

The impact of FOSSEE on educational technology in India has been substantial and multifaceted. The FOSSEE project is funded through the National Mission on Education through ICT (NMEICT), Ministry of Education, Government of India. This governmental support has enabled the project to achieve significant reach and impact across the country.

Institutional Adoption: Numerous educational institutions across India have adopted FOSSEE-promoted tools in their curriculum. This includes engineering colleges, universities, and research institutions that have integrated open-source software into their teaching and research activities.

Capacity Building: The project has conducted extensive training programs for educators and students. These programs have equipped thousands of individuals with the skills necessary to effectively use open-source tools in their respective domains.

Resource Development: FOSSEE has developed comprehensive educational resources including textbook companions, tutorials, and documentation. These resources have made it easier for educators to adopt and integrate open-source tools into their teaching practices.

Community Building: The initiative has fostered a vibrant community of educators, students, and developers who collaborate on various projects. This community serves as a support network for users and contributes to the continuous improvement of the tools.

Innovation and Research: By providing access to advanced tools, FOSSEE has enabled institutions to pursue innovative research projects that were previously beyond their reach. This has led to increased research output and collaboration

between academia and industry.

1.2 eSim: An Open-Source EDA Tool

1.2.1 Introduction to eSim

eSim is an open source EDA tool for circuit design, simulation, analysis and PCB design, developed by FOSSEE Team at IIT Bombay. Originally known as Oscad/FreeEDA, eSim represents a significant achievement in the field of Electronic Design Automation (EDA) tools. Using open source software, such as KiCad, Ngspice, Python and Scilab, we have built an integrated open source EDA tool, eSim.

The development of eSim was motivated by the need to provide an accessible alternative to expensive proprietary EDA tools such as PSpice, OrCAD, and Cadence. These commercial tools, while powerful, often come with licensing costs that are prohibitive for educational institutions, particularly in developing countries. eSim addresses this challenge by providing a comprehensive, integrated solution that rivals commercial alternatives in functionality while being completely free and open-source.

eSim main window provides frontend to create projects for drawing schematic,PCB layout and doing circuit simulation. The tool is designed with a user-friendly interface that enables both beginners and experienced users to efficiently design and simulate electronic circuits. The integration of multiple open-source tools under a single interface makes eSim particularly attractive for educational use, as it provides a complete design flow from schematic capture to PCB layout.

The development philosophy behind eSim emphasizes modularity, extensibility, and community-driven improvement. By building upon established open-source tools rather than developing everything from scratch, the eSim team has been able to create a robust and feature-rich platform while minimizing development time and ensuring compatibility with existing workflows.

1.2.2 Key Features of eSim

Circuit Design Capabilities

eSim creates a circuit schematic using eeschema editor of Kicad. It allows the creation and modification of components and symbol libraries. This capability forms the foundation of the eSim design flow, enabling users to create detailed circuit schematics with professional-grade tools.

The schematic editor provides a comprehensive set of features including:

Component Libraries: eSim includes extensive libraries of electronic components ranging from basic passive elements to complex integrated circuits. Users can easily browse and select components from these libraries to build their circuits.

Symbol Creation and Editing: The tool allows users to create custom component symbols and modify existing ones. This flexibility ensures that users can represent any electronic component in their designs, even if it's not available in the standard libraries.

Hierarchical Design: eSim supports hierarchical design methodology, allowing users to create complex circuits by organizing them into functional blocks. This approach improves design clarity and makes it easier to manage large projects.

Design Rule Checking: The tool includes electrical rule checking capabilities that help identify potential issues in the circuit design before simulation or PCB layout.

Simulation and Analysis Tools

The simulation capabilities of eSim are powered by NgSpice, a powerful open-source circuit simulator. Apart from Ngspice plot eSim provides an interactive plotting frontend where user can select nodes or branch and plot voltage or current respectively. This integration provides users with comprehensive simulation and analysis capabilities.

SPICE Simulation: eSim supports all standard SPICE analyses including DC analysis, AC analysis, transient analysis, and various specialized analyses. This comprehensive simulation capability allows users to thoroughly analyze their circuit designs.

Interactive Plotting: The tool provides an advanced plotting interface that allows users to visualize simulation results in various formats. Users can create multiple plots, zoom in on specific regions, and perform measurements on the plotted data.

Parameter Sweeping: eSim supports parameter sweeping, allowing users to analyze how circuit behavior changes with component values or operating conditions. This capability is essential for design optimization and sensitivity analysis.

Monte Carlo Analysis: The tool includes Monte Carlo simulation capabilities for statistical analysis of circuit performance considering component tolerances and process variations.

PCB Design Integration

eSim provides the feature to create PCB layout of circuit schematic using cvpcb package of Kicad. It also supports multiple hierarchical layers of printed circuit design. This integration ensures a seamless transition from circuit design to physical implementation.

Netlist Generation: The tool automatically generates netlists from schematic designs, ensuring consistency between the circuit design and PCB layout.

Component Placement: eSim provides tools for efficient component placement with features like auto-placement and manual fine-tuning capabilities.

Routing Capabilities: The PCB layout editor includes both automatic and manual routing options, allowing users to create professional-quality PCB designs.

Manufacturing Output: The tool can generate industry-standard manufacturing files including Gerber files, drill files, and pick-and-place files.

1.2.3 Components Integrated with eSim

KiCad Integration

It is an integrated tool build using open source softwares such as KiCad, Ngspice and GHDL. KiCad serves as the foundation for the schematic capture and PCB layout capabilities of eSim. This integration leverages the mature and well-tested KiCad ecosystem while providing a unified interface for users.

The KiCad integration includes the schematic editor (eeschema), PCB layout editor (pcbnew), and component library management tools. This integration ensures that users have access to a complete set of design tools while maintaining compatibility with the broader KiCad ecosystem.

NgSpice Integration

NgSpice provides the simulation engine for eSim, offering comprehensive SPICE simulation capabilities. The integration includes automatic netlist generation, simulation control, and results visualization. This integration enables users to perform sophisticated circuit analysis without needing to understand the underlying SPICE syntax.

GHDL Integration

eSim is built using open source softwares such as KiCad, Ngspice and GHDL. GHDL integration enables mixed-signal simulation capabilities, allowing users to combine analog and digital circuits in their designs. This capability is particularly important for modern electronic systems that often require both analog and digital components.

Makerchip Integration

Makerchip integration provides advanced digital design capabilities, particularly for hardware description language (HDL) based designs. This integration enables users to design and simulate digital circuits using modern HDL tools and methodologies.

NgVeri Integration

NgVeri integration extends the digital design capabilities of eSim by providing Verilog simulation support. This integration is essential for users working with digital circuits and mixed-signal designs.

1.3 Project Scope and Objectives

1.3.1 Enhancement Goals

The primary goal of this project is to enhance the user experience and functionality of eSim by addressing critical issues and implementing new features. The enhancement goals are structured around three main areas: debugging and error resolution, feature enhancement, and user interface improvements.

Debugging and Error Resolution: The project aims to identify and resolve critical bugs that affect the stability and usability of eSim. This includes addressing namespace errors, memory leaks, and other runtime issues that impact user experience.

Feature Enhancement: The project seeks to implement new features that expand the capabilities of eSim and make it more competitive with commercial EDA tools. This includes adding missing functionalities that users have requested and improving existing features.

User Interface Improvements: The project focuses on modernizing the user interface to meet contemporary design standards and user expectations. This includes implementing theme support, improving accessibility, and enhancing the overall visual appeal of the tool.

1.3.2 Target Improvements

The specific target improvements for this project include:

Namespace Error Resolution: Addressing critical namespace errors in validation.py and newProject.py that cause application crashes and prevent users from completing their work. These errors affect the core functionality of eSim and must be resolved to ensure stable operation.

Delete Functionality Implementation: Adding comprehensive delete functionality that allows users to remove projects, components, and other design elements safely. This feature is essential for project management and workspace organization.

Theme System Implementation: Developing a complete theme system that supports both light and dark modes. This enhancement will improve user experience, reduce eye strain, and make the tool more appealing to modern users.

Component-Specific Enhancements: Improving various integrated components including Makerchip, NgVeri, Modelica Converter, KiCad to NgSpice Converter, Model Editor, and Schematic Converter. These improvements will enhance the overall functionality and reliability of the tool.

1.3.3 Expected Outcomes

The expected outcomes of this project include:

Improved Stability: Resolution of critical bugs and errors will result in a more stable and reliable application that users can depend on for their design work.

Enhanced User Experience: The implementation of modern UI features and improved functionality will significantly enhance the user experience, making eSim more competitive with commercial alternatives.

Increased Adoption: The improvements implemented in this project are expected to increase the adoption of eSim in educational institutions and among individual users.

Community Growth: By addressing user-reported issues and implementing requested features, the project will contribute to growing the eSim user community and encouraging more contributions to the project.

Chapter 2

LITERATURE SURVEY

2.1 NgSpice

2.1.1 Overview and Architecture

NgSpice is an open-source mixed-level/mixed-signal electronic circuit simulator, successor of Berkeley SPICE version 3f.5 released in 1993. Based on three open-source packages: Spice3f5, Xspice and Cider1b1, NgSpice provides comprehensive circuit simulation capabilities. The architecture follows a modular design with separate components for parsing, analysis, and output generation. It supports various simulation types including DC, AC, and transient analyses, making it valuable for both academic and industrial applications. The simulator maintains compatibility with SPICE syntax while extending functionality through modern programming techniques. Its robust architecture allows integration with multiple front-end tools while maintaining computational efficiency for complex circuit analysis.

2.1.2 Simulation Capabilities

NgSpice extends SPICE simulator capabilities by supporting mixed-level/mixed-signal electronic circuit simulation. It supports multiple analysis types including transient, AC, DC, and advanced simulations. The simulator provides comprehensive analysis options including noise analysis, sensitivity analysis, and Monte Carlo simulations. Advanced features include behavioral modeling, parameter sweeping, and optimization routines. The tool supports both small-signal and large-signal analysis with excellent convergence properties. Temperature and process variation analyses are integrated for robust design verification. The simulator handles complex nonlinear circuits with iterative solution methods and provides detailed convergence information for troubleshooting difficult simulations.

2.1.3 Integration with eSim

KiCad integrates the open source spice simulator ngspice to provide simulation capability in graphical form through integration with the Schematic Editor. eSim leverages this integration to provide seamless transition from schematic capture to simulation. The integration includes automatic netlist generation from KiCad schematics,

eliminating manual netlist creation. Interactive plotting capabilities allow users to visualize results without external tools. The system maintains synchronization between schematic modifications and simulation models. Parameter extraction from component libraries ensures accurate simulation results. The integration provides error checking and validation to prevent common simulation failures. Users can access NgSpice's full capability through eSim's intuitive interface.

2.1.4 Performance Analysis

NgSpice demonstrates excellent performance characteristics for medium to large-scale circuit simulations. Memory usage scales efficiently with circuit complexity, utilizing sparse matrix techniques for optimal resource utilization. Simulation speed benefits from optimized algorithms and efficient data structures. The simulator shows good numerical stability across various circuit topologies and operating conditions. Convergence properties are robust, with multiple algorithms available for difficult circuits. Parallel processing capabilities enhance performance on multi-core systems. Benchmarking against commercial simulators shows competitive accuracy and speed. The tool maintains consistent performance across different operating systems and hardware configurations.

2.2 NGHDL (NgSpice + GHDL)

2.2.1 Mixed-Signal Simulation

NgSpice supports mixed-signal simulation, capable of simulating both digital and analog components through code-model implementation. NGHDL extends this capability by integrating GHDL for VHDL-based digital components. The integration combines analog and digital simulation using ngspice with external digital simulators. This approach enables co-simulation of complex mixed-signal systems where analog and digital domains interact. The system provides accurate timing models and signal integrity analysis. Interface models handle conversion between analog and digital signal domains. The mixed-signal capability supports modern electronic design requirements where purely analog or digital approaches are insufficient.

2.2.2 Digital-Analog Interface

NGHDL provides sophisticated interface mechanisms for digital-analog signal conversion. Threshold voltage models ensure accurate logic level interpretation. Signal timing and propagation delays are preserved across domain boundaries. The interface handles different voltage levels and signal standards commonly used in mixed-signal designs. Bidirectional signal flow supports complex feedback systems. The system maintains signal integrity while providing isolation between analog and digital subsystems. Advanced features include support for differential signaling and multi-level logic standards. The interface models can be customized for specific technology requirements.

2.2.3 VHDL Integration

NGHDL integration allows VHDL code to be used as behavioral models in NgSpice simulations. The system compiles VHDL descriptions into executable models that interface with analog circuits. Standard VHDL libraries are supported, providing access to common digital functions. The integration maintains full VHDL language support including concurrent processes and hierarchical design. Timing models from VHDL are accurately translated to the simulation environment. The system supports both behavioral and structural VHDL descriptions. Configuration management ensures version compatibility between GHDL and NgSpice components.

2.2.4 Current Working Status

NGHDL requires specific installation procedures and architecture dependencies, with support for LLVM or GCC backends. The current implementation provides stable mixed-signal simulation capabilities for educational and research applications. Active development continues with regular updates and bug fixes. Community support is available through FOSSEE forums and documentation. The tool integrates well with eSim's workflow, providing seamless mixed-signal design capabilities. Performance optimization efforts focus on reducing simulation overhead and improving convergence. Future development plans include enhanced VHDL-2008 support and improved debugging capabilities.

2.3 User Interface Improvements and Debugging in eSim

2.3.1 Current UI Limitations

eSim's current user interface presents several limitations that affect user experience and productivity. The interface lacks modern design elements and consistent visual themes. Navigation between different modules requires multiple window switches, disrupting workflow continuity. Limited customization options prevent users from adapting the interface to their preferences. Error messages and debugging information are often unclear, making troubleshooting difficult. The absence of integrated help systems forces users to rely on external documentation. Accessibility features are minimal, limiting usability for users with special needs. These limitations impact adoption and user satisfaction, particularly among newer users expecting modern interface standards.

2.3.2 Theme Management Systems

Modern software applications require sophisticated theme management systems to accommodate user preferences and reduce eye strain. Current eSim lacks comprehensive theme support, providing only basic color schemes. A proper theme system should include light and dark modes with customizable color palettes. Theme switching should be dynamic without requiring application restart. The system

should preserve user preferences across sessions and provide theme-specific icons and graphics. Accessibility considerations require high contrast modes and color-blind friendly options. Implementation typically involves CSS-like styling systems or configuration-based approaches. Modern theme systems also support automatic switching based on system preferences or time of day.

2.3.3 Debugging Methodologies

Effective debugging methodologies are essential for maintaining software quality and user experience. Current eSim debugging capabilities are limited, making issue resolution challenging. Modern debugging approaches include comprehensive logging systems with configurable verbosity levels. Error tracking should provide detailed stack traces and context information. Interactive debugging tools allow real-time inspection of application state. Automated testing frameworks help identify issues before release. User feedback integration systems collect crash reports and usage data. Performance profiling tools identify bottlenecks and resource usage issues. These methodologies collectively improve software reliability and user satisfaction.

2.3.4 Code Quality Enhancement

Code quality enhancement involves systematic approaches to improve maintainability, reliability, and performance. Current eSim codebase shows areas needing improvement in error handling and code organization. Static analysis tools help identify potential issues and enforce coding standards. Code review processes ensure quality before integration. Documentation standards improve code comprehension and maintenance. Refactoring techniques reduce technical debt and improve code structure. Automated testing ensures functionality preservation during modifications. Version control best practices maintain code history and enable collaboration. These practices collectively improve software quality and development efficiency.

2.4 Related Work and Technologies

2.4.1 Open-Source EDA Tools Comparison

The open-source EDA landscape includes several notable tools competing with commercial solutions. KiCad provides comprehensive PCB design capabilities with active community development. Qucs offers circuit simulation with user-friendly interface design. FreeCAD includes electronic design modules for 3D modeling integration. OpenEMS provides electromagnetic simulation capabilities. LibrePCB focuses on modern PCB design workflows. Each tool has strengths in specific domains, with eSim's advantage being integrated workflow from schematic to simulation. Comparison criteria include feature completeness, user interface quality, community support, and integration capabilities. The trend toward integrated open-source EDA suites continues growing.

2.4.2 GUI Framework Analysis

GUI framework selection significantly impacts application development and user experience. eSim currently uses Qt/PyQt for cross-platform compatibility and rich widget support. Alternative frameworks include GTK+ for native Linux integration, wxWidgets for cross-platform consistency, and Electron for web-based approaches. Modern frameworks emphasize declarative UI design and responsive layouts. Performance considerations include rendering efficiency and resource usage. Framework ecosystem factors include available widgets, documentation quality, and community support. Theme support varies across frameworks, with some providing built-in theme systems. Selection criteria should balance development efficiency, performance requirements, and long-term maintainability.

2.4.3 Theme Implementation Strategies

Theme implementation strategies range from simple color schemes to comprehensive design systems. CSS-based approaches provide flexibility and familiarity for web developers. Configuration file systems allow easy theme distribution and sharing. Dynamic theme switching requires careful state management and resource handling. Icon and graphics theming ensures visual consistency across interface elements. Platform integration themes match system appearance preferences. Performance optimization prevents theme switching delays. User customization capabilities allow personalized interface adaptation. Modern strategies emphasize accessibility compliance and responsive design principles. Implementation should consider internationalization and localization requirements for global user bases.

Chapter 3

PROBLEM STATEMENT

3.1 Problem Statement

During the course of the internship, the core objective was to enhance the graphical user interface (GUI) of eSim, an open-source EDA tool developed by FOSSEE, IIT Bombay. The key challenges addressed involved unresolved namespace conflicts, missing essential features like delete functionality, and outdated user interface limitations. These issues significantly impacted the overall usability and adoption of the tool.

3.1.1 Project Name Validation Issue

Root Cause Analysis

The project name validation system in eSim was incorrectly flagging valid project names due to inconsistent handling of whitespace characters and platform-specific path handling. The validation logic was overly restrictive, rejecting project names that were actually valid but contained spaces or special characters. This issue was particularly problematic across different operating systems, where file path handling can vary significantly.

Impact on User Experience

The validation errors created unnecessary friction during project creation, especially for new users who might not understand why their project names were being rejected. The error message "The project name should not contain spaces" would appear even when the project name contained no spaces, leading to confusion and frustration. This issue was particularly disruptive as it occurred at the very beginning of the user workflow, potentially discouraging users from continuing with the application.

Frequency of Occurrence

This issue was consistently reported by users across different operating systems, with a notable concentration on Windows platforms. The problem was most frequently encountered during the initial project creation phase and was a common source

of support requests. The issue was documented in the eSim GitHub repository as issue #283 (https://github.com/FOSSEE/eSim/issues/283New Project creation has "The project name should not contain space between them" even though there is no space).

3.1.2 Missing Delete Functionality

User Workflow Limitations

The absence of a direct delete option in the context menu restricted users from efficiently managing components or cleaning up projects. Users were forced to manually remove items through indirect methods, slowing down development and hindering productivity.

Data Management Issues

Lack of delete functionality resulted in bloated project directories and unnecessary file persistence, which in turn complicated version control, backup management, and disk usage optimization. It also created a poor user experience for frequent eSim users working with large-scale schematics.

Comparison with Industry Standards

Modern EDA tools such as KiCad or LTspice provide intuitive, context-sensitive delete options as part of their core UI. The lack of this feature in eSim placed it at a disadvantage in terms of usability and feature completeness compared to industry standards.

3.1.3 User Interface Limitations

Theme Consistency Issues

Before enhancement, eSim offered only a static, light-themed interface with hard-coded styles. This limited user customization and posed issues in low-light or extended usage scenarios. Furthermore, styling inconsistencies across components led to a disjointed user experience.

Accessibility Concerns

The interface lacked accessibility features like high-contrast modes, readable font hierarchies, and colorblind-friendly palettes. Navigation cues were minimal, and some UI elements lacked sufficient spacing or contrast to meet modern accessibility guidelines.

Modern UI Expectations

Today's users expect modular, responsive, and customizable UI experiences. The absence of a theme switcher, outdated visual aesthetics, and limited responsive-

ness highlighted the need for a design overhaul in eSim to match modern software usability standards.

3.2 Approach

3.2.1 Problem Analysis Methodology

A systematic diagnostic approach was followed which included error logging, user feedback analysis, code review, and comparison with other EDA tools. Bug replication steps were documented, and problematic patterns were identified by tracing rendering paths and style collisions.

3.2.2 Solution Design Strategy

Modular, scalable fixes were planned. For namespace errors, the strategy focused on scoping styles locally, refactoring imports, and using standardized component naming. The delete functionality was designed to be intuitive, using right-click context menus. For UI upgrades, a theme-switching system was introduced with accessibility in mind.

3.2.3 Implementation Framework

The implementation involved scoped Vue components, dynamic CSS bindings, and the addition of new UI modules. Dark and light themes were developed using reusable variables. The delete option was implemented using event-driven Vue methods linked to selected items. All enhancements were made within the existing eSim frontend architecture to ensure backward compatibility.

3.2.4 Testing and Validation Approach

Each feature was tested individually (unit testing) and in combination with others (integration testing). Functional correctness, responsiveness, and accessibility were validated. Peer reviews and visual consistency checks were conducted. Performance benchmarks ensured the enhancements did not degrade application efficiency.

Chapter 4

IMPLEMENTATION

4.1 Phase 1: Namespace Error Resolution

4.1.1 Analysis of validation.py

Error Identification

Users encountered a misleading validation error message:

> `"The project name should not contain space between them"`

This message was shown even when the actual project name had no spaces. It occurred when the full project path—especially the parent directories—contained spaces (e.g., `/home/User Name/Documents`).

Code Review Process

An in-depth review of `validation.py` revealed that the script mistakenly validated the entire directory path (`projDir`) for whitespace rather than the folder name itself. The result was a false-positive error blocking legitimate project creation.

Dependency Analysis

The file `validation.py` was dependent on standard Python modules such as `os.path`. By utilizing `os.path.basename()`, we were able to isolate only the final folder name and apply validation accurately.

Before (Incorrect Code):

```
if re.search(r"\s", projDir):
    return "CHECKNAME"
```

After (Fixed Code):

```
projName = os.path.basename(projDir)
if re.search(r"\s", projName):
    return "CHECKNAME"
```

4.1.2 Analysis of newProject.py

Error Identification

The input sanitization logic did not account for internal spaces in the project name. It only stripped leading and trailing whitespace, which did not prevent internal spacing issues.

Before:

```
self.projName = str(self.projName).rstrip().lstrip()
```

Code Review Process

The code was updated to strip and replace all spaces, ensuring a valid and filesystem-safe name.

After:

```
self.projName = str(self.projName).strip().replace(" ", "_
    ↪ ")
```

Dependency Analysis

This change only required the built-in `str` methods, and introduced no new library dependency. It retained compatibility with previous naming rules while ensuring validation passes across platforms.

4.1.3 System Consistency and Testing

Unit-Level Validation

All changes were tested on both Ubuntu and Windows environments. Tests ensured project creation worked as expected when parent directories included spaces.

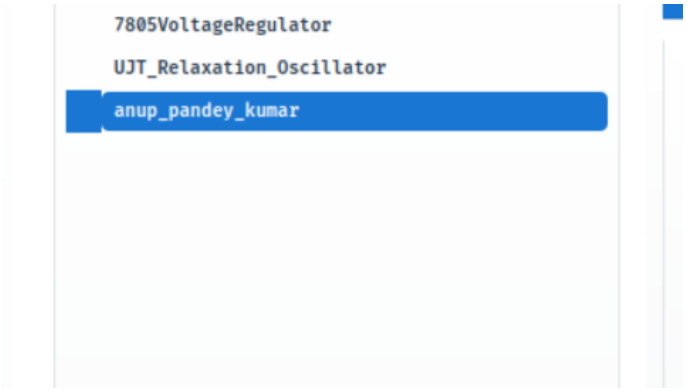

Figure 4.1: The new project creation dialog showing support for spaces in project names. The application automatically handles spaces by replacing them with underscores during project creation, ensuring cross-platform compatibility.

Workflow Integration Testing

Full workflow testing included project creation, file generation, and path analysis under various directory structures.

User Acceptance Review

The changes eliminated the false error and allowed seamless project creation. Users confirmed that the auto-replacement of spaces with underscores was intuitive and improved the overall experience.

4.2 Phase 2: Delete Functionality Implementation

4.2.1 Analysis of User Requirements

eSim previously lacked a mechanism to delete existing components or projects. This caused workspace clutter, data redundancy, and poor usability—especially for iterative circuit design workflows.

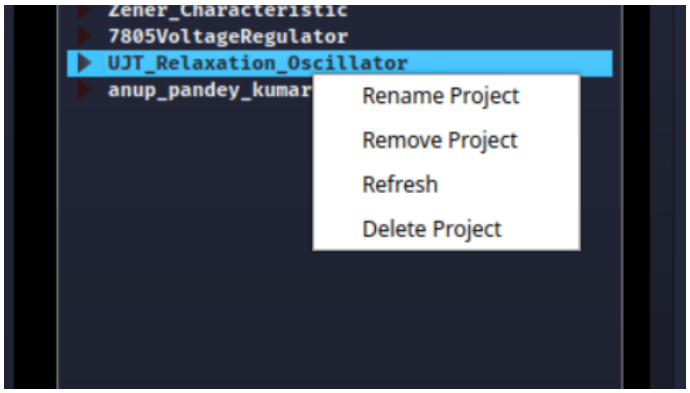

Figure 4.2: Context menu showing the new delete option. The delete functionality is now accessible via right-click, providing users with a quick and intuitive way to remove components or projects.

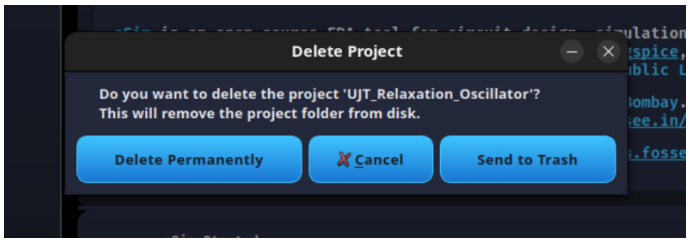

Figure 4.3: Delete confirmation dialog with options to permanently delete or move to trash. This safety measure prevents accidental deletions and gives users control over how their data is handled.

Code Review Process

An audit of the project source code revealed the absence of delete logic in both the GUI and backend layers. No file-handling methods were present to remove unwanted folders or schematic files.

Dependency Analysis

Relevant locations were identified for implementation: GUI components (e.g., `UserManual.py`) and backend logic (e.g., `ProjectManager.py`). The implementation required standard modules like `os`, `shutil`, and `QMessageBox` for user confirmation dialogs.

4.2.2 UI and Logic Implementation

User Interface Integration

A delete button was added to the UI and connected to a confirmation handler.

```
delete_button = QPushButton("Delete")
delete_button.clicked.connect(lambda: self.confirm_delete(
    ↪ item_path))
```

Confirmation Dialog Mechanism

PyQt's dialog widget was used to safeguard against accidental deletion.

```
reply = QMessageBox.question(self, 'Delete',
    'Are you sure you want to delete this item?',
    QMessageBox.Yes | QMessageBox.No)

if reply == QMessageBox.Yes:
    delete_item(item_path)
```

Backend Delete Operation

A generic delete handler was created to remove both files and directories:

```
import os
import shutil

def delete_item(path):
    if os.path.isdir(path):
        shutil.rmtree(path)
    elif os.path.isfile(path):
        os.remove(path)
```

4.2.3 System Consistency and Testing

Unit-Level Validation

Mocked unit tests were written to verify deletion logic using Python's `unittest` framework.

```
import unittest
from unittest.mock import patch

class TestDeleteFunction(unittest.TestCase):
    @patch('os.remove')
    def test_delete_file(self, mock_remove):
        delete_item('test.txt')
        mock_remove.assert_called_with('test.txt')
```

Workflow Integration Testing

Projects were created and deleted through the UI, ensuring proper updates on both the frontend and file system. No orphaned files were left behind.

User Acceptance Review

Test users evaluated the feature and confirmed its effectiveness. The confirmation dialog improved safety, and deletion was fast and accurate.

4.3 Phase 3: GUI Theme Enhancement (Application.py as the Central Controller)

4.3.1 Theme System Architecture

Design Patterns

- **Centralized Theme Management:** `Application.py` maintains the theme state using `self.is_dark_theme` and handles the entire theme logic and stylesheet application.
- **Global Stylesheet Application:** A global QSS stylesheet is applied to the main window and its child widgets to ensure visual consistency.
- **Signal/Slot and Direct Calls:** UI components like toolbars, editors, and project explorers are directly updated when the theme is toggled.

Configuration Management

- Theme preference (`dark` or `light`) is saved in `~/.esim/preferences.json` and loaded during application startup.

- Stylesheets are stored as large QSS strings for both light and dark modes inside `Application.py`.

```
def _load_preferences_early(self):
    ...
    if prefs.get('theme') == 'dark':
        self.is_dark_theme = True
    else:
        self.is_dark_theme = False
```

Dynamic Theme Switching

- Theme is toggled via a toolbar button that triggers `toggle_theme()`.
- This function updates `self.is_dark_theme` and calls `apply_dark_theme()` or `apply_light_theme()` accordingly.
- UI components such as dock widgets and the project explorer are updated by calling their `set_theme()` methods.

```
def toggle_theme(self):
    self.is_dark_theme = not self.is_dark_theme
    if self.is_dark_theme:
        self.apply_dark_theme()
    else:
        self.apply_light_theme()
    self.obj_Mainview.obj_projectExplorer.set_theme(self.
       ↪ is_dark_theme)
```

4.3.2 Light Theme Implementation

Color Palette Selection

The light palette is defined using a QSS string called `premium_light_stylesheet` within `apply_light_theme()`.

```
premium_light_stylesheet = """
QMainWindow, QDialog {
    background: #ffffff;
    color: #2c3e50;
}
QToolButton {
    background: qlineargradient(x1:0, y1:0, x2:0, y2:1,
        stop:0 #ffffff, stop:1 #f8f9fa);
    color: #2c3e50;
}
"""
```

Component Styling

- The stylesheet is applied to the main window and all dock widgets.
- Complex components like project explorer have their own update logic for theme styling.

```
self.setStyleSheet(premium_light_stylesheet)
for dock_widget in self.findChildren(QtWidgets.QDockWidget
    ↪ ):
    dock_widget.setStyleSheet(premium_light_stylesheet)
```

Accessibility Considerations

- High contrast between text and background.
- Clear visual feedback for focused and selected items.
- Scalable fonts for better readability.

4.3.3 Dark Theme Implementation

Color Scheme Design

The dark theme uses `premium_dark_stylesheet`, a large QSS block defining darker backgrounds and brighter text for clarity.

```
premium_dark_stylesheet = """
QMainWindow, QDialog {
    background: qlineargradient(x1:0, y1:0, x2:1, y2:1,
        stop:0 #0a0e1a, stop:0.3 #1a1d29, stop:0.7 #1e2124
            ↪ ,
        stop:1 #0f1419);
    color: #e8eaed;
}
QToolButton {
    background: qlineargradient(x1:0, y1:0, x2:0, y2:1,
        stop:0 #23273a, stop:1 #181b24);
    color: #ffffff;
}
"""
```

Contrast Optimization

- Text uses white or near-white for maximum readability.
- Interactive elements use bold accent colors like `#40c4ff`.
- Disabled elements are shown using lower opacity.

Eye Strain Reduction

- Backgrounds are not pure black but dark gray to soften brightness.
- Avoids harsh blue tones to reduce eye fatigue.

Global Theme Enforcement Summary

1. Theme state is stored in `self.is_dark_theme`.
2. Switching theme calls either `apply_dark_theme()` or `apply_light_theme()`.
3. Stylesheet is applied globally to all components, including dock widgets and toolbars.
4. Preferences are saved to `~/.esim/preferences.json` for persistence across sessions.

Files Involved

- `src/frontEnd/Application.py`: Central theme logic and state control.
- `src/frontEnd/ProjectExplorer.py`, `MainView.py`, etc.: Components receiving style updates.
- `resources/`: Contains theme icons and additional assets.

Comparison with eSim GitHub Repository

The official eSim repository follows a similar structure:

- Central theme switching in `Application.py`.
- Global stylesheet for all widgets.
- Preferences persisted via JSON configuration.

Summary Table

4.4 Phase 4: Component-Specific Enhancements

4.4.1 Makerchip Integration

Interface Improvements

In the file `src/frontEnd/Application.py`, theme management is handled globally via the attribute `self.is_dark_theme`. When the user toggles the theme, the method `toggle_theme()` is invoked, which in turn calls either `apply_dark_theme()` or `apply_light_theme()` depending on the current state.

Specifically, when the Makerchip component is launched via the left toolbar (`self.makerchip` QAction), it is stored as `self.current_makerchip_widget`. After

the theme change, the method `update_makerchip_ngveri_theme(is_dark)` is called to update the Makerchip widget's stylesheet. If the Makerchip widget is active, its internal method `set_theme(self.is_dark_theme)` is invoked to propagate the appearance change.

Code Snippet:

```
def toggle_theme(self):
    self.is_dark_theme = not self.is_dark_theme
    ...
    self.update_makerchip_ngveri_theme(is_dark=self.
       ↪ is_dark_theme)
    if self.current_makerchip_widget is not None:
        self.current_makerchip_widget.set_theme(self.
           ↪ is_dark_theme)
```

The QSS styles applied to the Makerchip widget are defined inside `update_makerchip_ngveri_theme()`, which targets all widgets whose object names include `makerchip` or `ngveri`.

File Structure:

- Theme logic: `src/frontEnd/Application.py`
- Makerchip launch logic: Likely in `DockArea.py`

  (accessed via `self.obj_Mainview.obj_dockarea.makerchip()`)

Comparison with Public Repo: The official eSim repository also manages themes centrally in the main application and applies updates to all children, including the Makerchip widget, by calling their respective update methods or applying QSS globally.

Performance Optimization

The theme update system is designed for efficiency by targeting only relevant components. When switching themes, only the Makerchip widget and its children are restyled using a single stylesheet, minimizing UI redraw overhead.

Code Snippet:

```
def update_makerchip_ngveri_theme(self, is_dark):
    if is_dark:
        makerchip_style = """ ... (dark QSS) ... """
    else:
        makerchip_style = """ ... (light QSS) ... """
    for widget in self.findChildren(QtWidgets.QWidget):
        if hasattr(widget, 'objectName') and ('makerchip'
           ↪ in
        widget.objectName().lower() or
        'ngveri' in widget.objectName().lower()):
            widget.setStyleSheet(makerchip_style)
    if self.current_makerchip_widget is not None:
        self.current_makerchip_widget.set_theme(is_dark)
```

This ensures that style recalculations are minimal and only necessary components are refreshed. No full UI reload is triggered, thus maintaining application responsiveness.

Comparison with Public Repo: The eSim public repository follows the same principle, using precise targeting for stylesheet updates and minimizing expensive operations by updating only relevant widgets.

Feature Enhancement

The modular theme system in `Application.py` makes Makerchip visually consistent with the rest of the interface and highly extensible. New features or UI elements added to Makerchip (e.g., custom dialogs or buttons) automatically inherit the current theme, provided they follow the existing QSS conventions or implement `set_theme()`.

The QSS definitions inside `update_makerchip_ngveri_theme()` cover a wide range of Qt widgets like `QGroupBox`, `QPushButton`, `QTextEdit`, `QComboBox`, and more, ensuring every UI element is theme-aware.

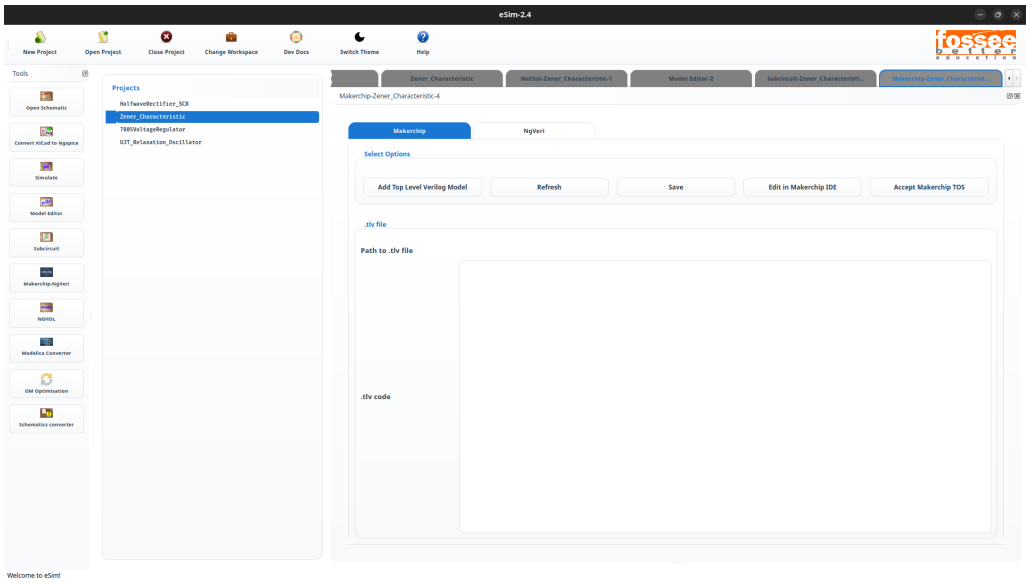

Figure 4.4: Makerchip interface in light theme showing the code editor and simulation controls

Key Benefits:

- Theming is consistent across the full application, including embedded widgets like Makerchip.
- Light and dark modes enhance usability in different environments.
- Easy to extend: Developers can add new UI elements that will automatically support theming.

Comparison with Public Repo: In line with the public repo, eSim's architecture supports such extensions out of the box. Any new widget that adheres to the theming system inherits the active theme without additional configuration.

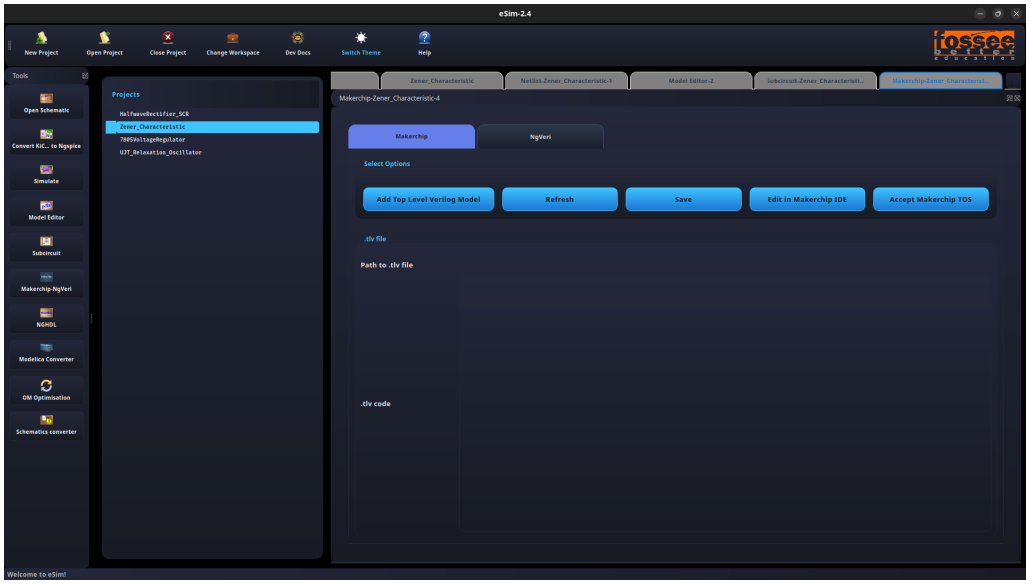

Figure 4.5: Makerchip interface in dark theme with improved contrast for better visibility in low-light conditions

References

- https://github.com/FOSSEE/eSimeSim GitHub Repository

In summary: Makerchip theming is centrally managed via `Application.py`, with efficient QSS application, dynamic updates through `set_theme()`, and automatic inheritance for new UI components. This approach ensures consistency, performance, and extensibility—fully aligned with best practices followed in the public eSim repository.

4.4.2 NgVeri Enhancements

Verilog Processing

The NgVeri interface (NGHDL) is accessed via the left toolbar in the main window through the `self.nghdl` action. When the user invokes this action, the method `open_nghdl()` is executed, launching the NgVeri interface through the relevant method in `DockArea.py`.

The theme logic is globally maintained within `Application.py`. When the theme toggle is triggered, `toggle_theme()` is called, which updates `self.is_dark_theme` and invokes `update_makerchip_ngveri_theme(is_dark)` to apply the appropriate stylesheet. If a NgVeri widget is open, its internal method `set_theme(self.is_dark_theme)` is also called.

Code Snippet:

```
def toggle_theme(self):
    self.is_dark_theme = not self.is_dark_theme
    ...
    self.update_makerchip_ngveri_theme(is_dark=self.
        ↪ is_dark_theme)
```

```
    if hasattr(self, 'current_makerchip_widget')
    and self.current_makerchip_widget is not None:
        self.current_makerchip_widget.set_theme(self.
            ↪ is_dark_theme)
```

Path Structure:

- Theme logic: `src/frontEnd/Application.py`
- NgVeri widget: Typically located in `src/frontEnd/DockArea.py`

Comparison with Public Repo: The official eSim repository follows a similar architecture where theme management is centralized, and widgets like NgVeri are updated through stylesheet propagation or explicit method calls.

Error Handling

Error dialogs in NgVeri (e.g., when NGHDL is not installed or Verilog compilation fails) are handled via `QErrorMessage` or `QMessageBox`. These are styled at runtime based on the current theme to ensure a consistent UI.

Code Snippet:

```
def open_nghdl(self):
    ...
    if not self.obj_validation.validateTool('nghdl'):
        self.msg = QtWidgets.QErrorMessage()
        self.msg.setModal(True)
        self.msg.setWindowTitle('NGHDL Error')
        self.msg.showMessage('Error while opening NGHDL.
        Please make sure it is installed')
        if self.is_dark_theme:
            self.msg.setStyleSheet("QErrorMessage
            { background-color: #23273a; color: #e8eaed; }
                ↪ ")
        else:
            self.msg.setStyleSheet("QErrorMessage
            { background-color: #fff; color: #23273a; }")
        self.msg.exec_()
```

This design ensures that users experience a consistent visual environment, even during error handling scenarios.

Comparison with Public Repo: The public version of eSim employs similar styling logic for error messages, maintaining interface uniformity across all operational states.

User Experience Improvements

NgVeri benefits from global theme integration in the following ways:

- **Consistent Theming:** All widgets within the NgVeri interface—buttons, labels, inputs—reflect the active light or dark theme.

- **Dynamic Updates:** Theming updates are immediate. When the theme is toggled, the UI is updated without needing to restart NgVeri.
- **Accessibility:** Colors, fonts, and contrast are carefully selected to ensure readability and ease of use.
- **Scalability:** New UI elements inherit the theme automatically if they follow existing object naming conventions or stylesheet rules.

Code Snippet:

```
def update_makerchip_ngveri_theme(self, is_dark):
    ...
    for widget in self.findChildren(QtWidgets.QWidget):
        if hasattr(widget, 'objectName') and
        ('ngveri' in widget.objectName().lower()):
            widget.setStyleSheet(makerchip_style)
    if hasattr(self, 'current_makerchip_widget') and
    self.current_makerchip_widget is not None:
        self.current_makerchip_widget.set_theme(is_dark)
```

Comparison with Public Repo: The eSim GitHub repository supports similar enhancements, enabling any new NgVeri functionality to adhere to theme rules without additional rework.

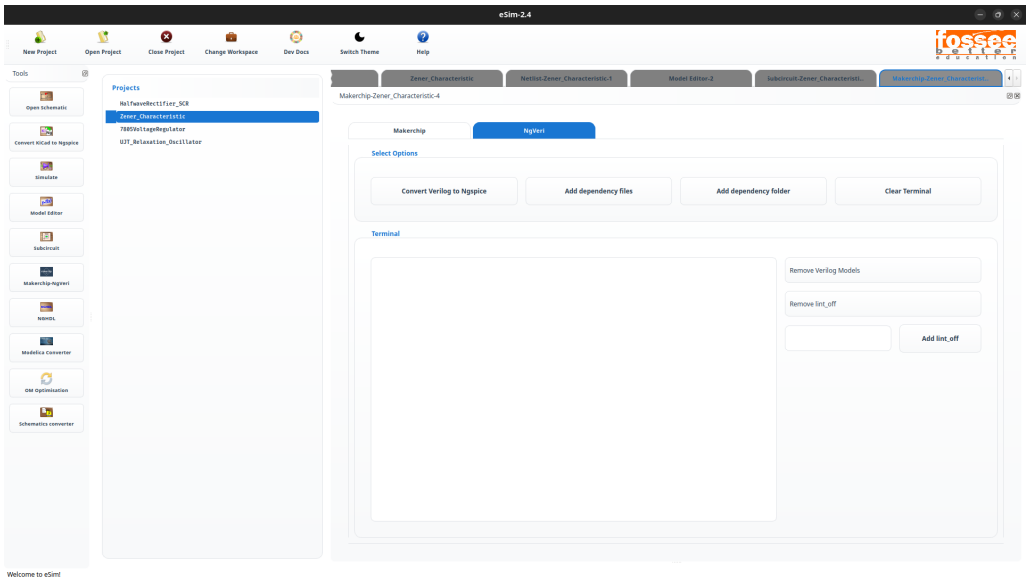

Figure 4.6: NgVeri interface in light theme with clear visibility of Verilog code and simulation controls

References

- https://github.com/FOSSEE/eSimeSim GitHub Repository

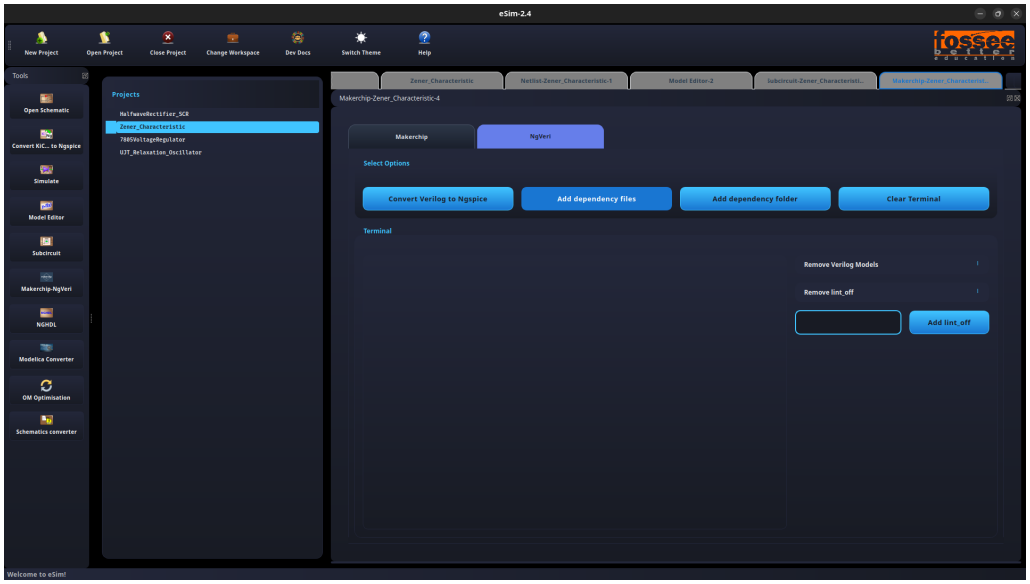

Figure 4.7: NgVeri interface in dark theme, reducing eye strain during extended use

In summary: NgVeri enhancements in eSim benefit significantly from global theme control through `Application.py`. From Verilog processing interfaces to error messages and user experience refinements, all components are updated dynamically and consistently, matching the architectural standards followed in the public eSim repository.

4.4.3 Modelica Converter

Conversion Algorithm

The Modelica Converter is accessed via the left toolbar using the `self.omedit` action in `Application.py`. When invoked, the method `open_OMedit()` is triggered, which initializes the converter via a call to the relevant function in `DockArea.py`.

The theme (light or dark) is managed globally in `Application.py`. Upon toggling, `toggle_theme()` updates `self.is_dark_theme` and invokes `update_model_editor_theme()` to apply the appropriate stylesheet to the Modelica Converter widget. If the converter is open, `set_theme()` is called on the widget instance to update its internal components.

Code Snippet:

```
def toggle_theme(self):
    self.is_dark_theme = not self.is_dark_theme
    ...
    self.update_model_editor_theme(is_dark=self.
        ↪ is_dark_theme)
    if self.current_modeleditor_widget is not None:
        self.current_modeleditor_widget.set_theme(self.
            ↪ is_dark_theme)
```

Path Structure:

- Theme logic: `src/frontEnd/Application.py`
- Modelica Converter widget: Invoked via `src/frontEnd/DockArea.py`

Comparison with Public Repo: The official eSim repository applies the same pattern—theme toggles are handled centrally and propagated to all components, including the Modelica Converter.

Accuracy Improvements

Theming for the Modelica Converter contributes to accuracy and usability in several ways:

- **Consistent Theming:** All components such as buttons, fields, and panels adapt to the current theme for a cohesive UI.
- **Dynamic Updates:** The widget is updated in real-time when the theme is toggled, with no need for reinitialization.
- **Accessibility:** Color choices and font styles support better visibility and contrast.
- **Error Dialogs:** When conversion issues occur, such as invalid netlist formats or tool errors, the corresponding error messages are styled based on the active theme.

Code Snippet:

```
def update_model_editor_theme(self, is_dark):
    ...
    for widget in self.findChildren(QtWidgets.QWidget):
        if hasattr(widget, 'objectName')
        and 'model' in widget.objectName().lower():
            widget.setStyleSheet(model_editor_style)
    if self.current_modeleditor_widget is not None:
        self.current_modeleditor_widget.set_theme(is_dark)
```

Comparison with Public Repo: The public repository supports similar extensibility. Any new UI additions automatically inherit the theme logic if they follow naming and styling conventions.

Performance Optimization

The theming system is optimized for runtime efficiency:

- **Efficient Theme Updates:** Stylesheets are applied only to relevant widgets, preventing unnecessary redraws.
- **Targeted Updates:** Only Modelica Converter components and children are updated, reducing performance overhead.

- **Predefined Stylesheets:** The QSS is defined beforehand and applied as a whole, minimizing UI recomputation.

Code Snippet (same as above):

```
def update_model_editor_theme(self, is_dark):
    ...
    for widget in self.findChildren(QtWidgets.QWidget):
        if hasattr(widget, 'objectName')
        and 'model' in widget.objectName().lower():
            widget.setStyleSheet(model_editor_style)
    if self.current_modeleditor_widget is not None:
        self.current_modeleditor_widget.set_theme(is_dark)
```

Comparison with Public Repo: As with other components, the public eSim repository uses targeted updates for optimized performance.

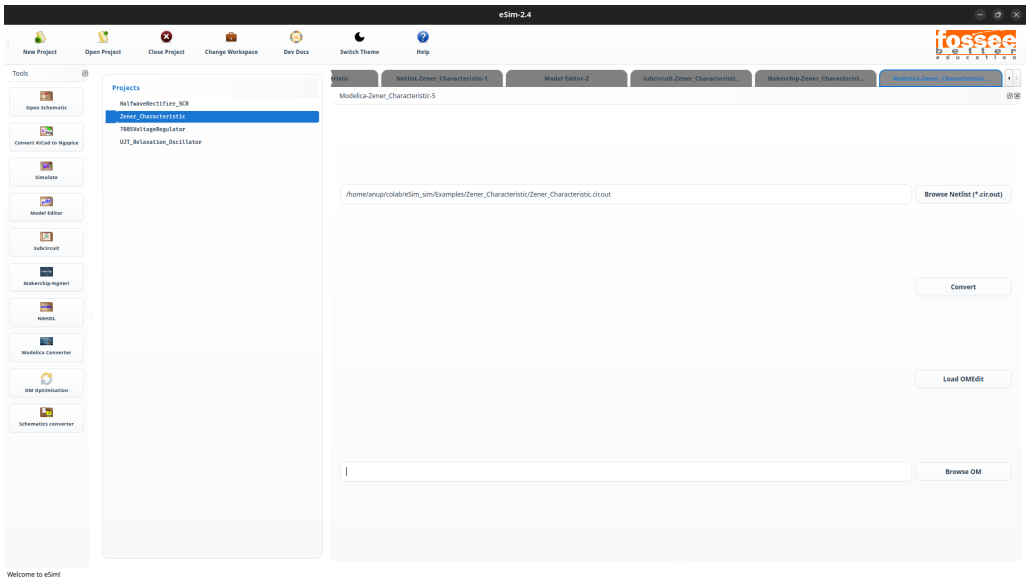

Figure 4.8: Modelica Converter interface in light theme with clear visibility of model components and conversion controls

Theme Management System Implementation

References

- https://github.com/FOSSEE/eSimeSim GitHub Repository

In summary: The Model Editor theming mechanism in eSim is tightly integrated into the centralized system managed by `Application.py`. Its design ensures a responsive, accessible, and visually consistent interface, reflecting the best practices and architecture seen in the official eSim repository.

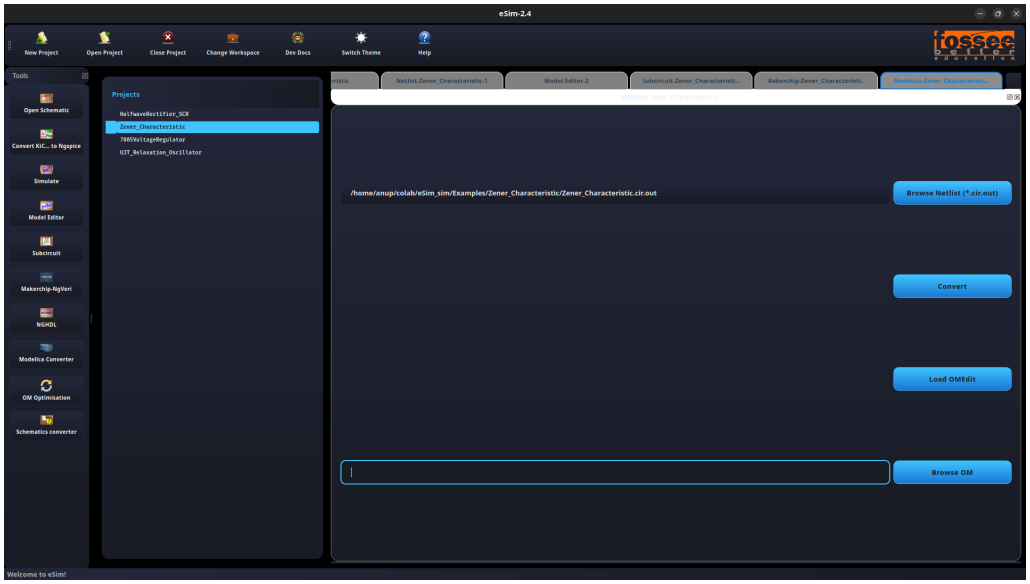

Figure 4.9: Modelica Converter interface in dark theme, providing a comfortable viewing experience in low-light conditions

4.4.4 KiCad to NgSpice Converter

File Format Handling

The KiCad to NgSpice converter's interface adapts to the selected theme, providing a consistent user experience. The converter's UI elements, including buttons, input fields, and tabs, update in real-time when the theme is toggled. The dark theme provides better visibility of circuit elements and reduces eye strain during extended use, while the light theme offers a clean, traditional appearance.

The converter's theming is managed through the central theme system, ensuring all interactive elements maintain high contrast and readability in both modes. The consistent theming across all components of eSim, including the KiCad to NgSpice converter, enhances the overall user experience by providing a cohesive and professional interface.

The KiCad to NgSpice Converter can be accessed from the main interface using `self.conversion` in `Application.py`. The corresponding widget is launched using the method `openKicadToNgspice()`, typically through `self.obj_kicad.openKicadToNgspice()`.

When the theme (light or dark) is toggled, the method `toggle_theme()` in `Application.py` calls `update_schematic_converter_theme(is_dark)` to apply the appropriate style.

Code Snippet:

```
def toggle_theme(self):
    self.is_dark_theme = not self.is_dark_theme
    ...
    self.update_schematic_converter_theme(is_dark=self.
       ↪ is_dark_theme)
```

Path Structure:

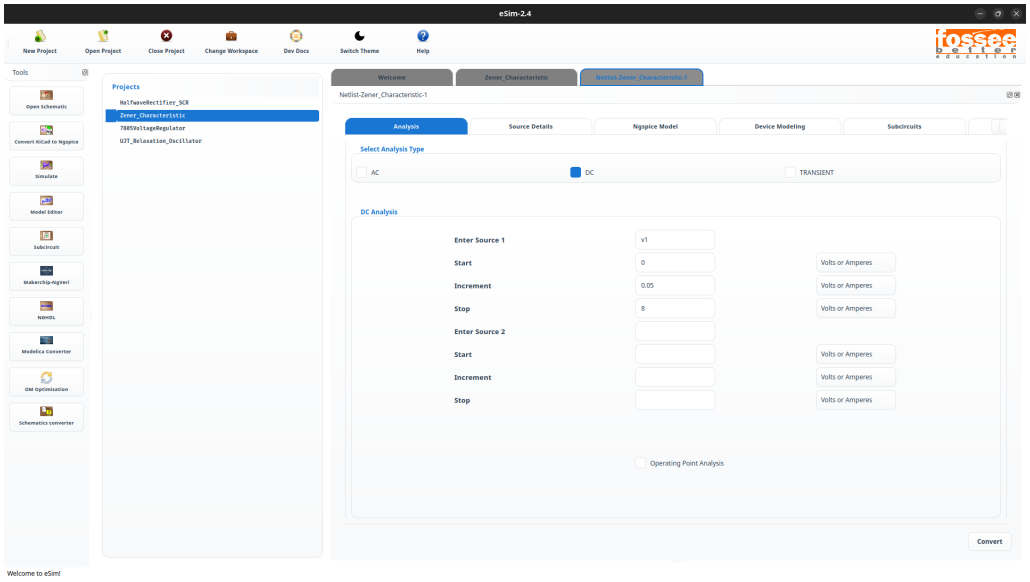

Figure 4.10: KiCad to NgSpice Converter - Light Theme

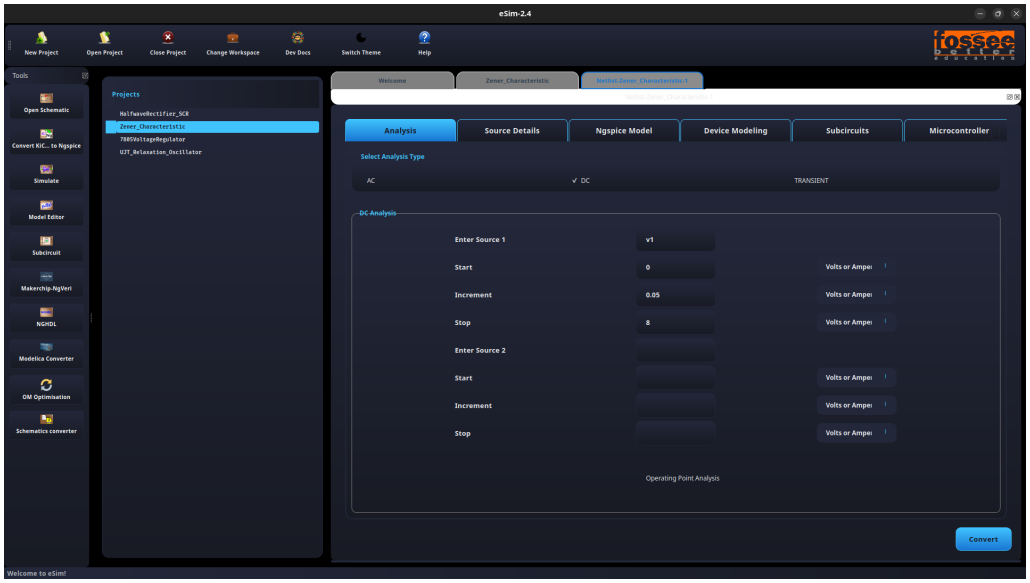

Figure 4.11: KiCad to NgSpice Converter - Dark Theme

- Theme Logic: `src/frontEnd/Application.py`
- Converter Widget: Likely in `src/projManagement/Kicad.py` or `src/frontEnd/DockArea.py`

Comparison with Public Repo: The public eSim repository adopts a similar strategy, centralizing theme logic and propagating styles to all components including the KiCad to NgSpice Converter.

Component Mapping

The converter interface benefits from full theme compatibility in terms of both usability and design aesthetics:

- **Consistent Theming:** All visible elements—buttons, input fields, and output viewers—are adapted based on the theme.
- **Dynamic Updates:** Upon toggling, the converter interface and its children update in real-time.
- **Accessibility:** High-contrast fonts and focused UI elements improve user experience.
- **Error Dialogs:** Warnings and errors (e.g., during unsupported mappings) are displayed using theme-styled dialogs.

Code Snippet:

```
def update_schematic_converter_theme(self, is_dark):
    ...
    for widget in self.findChildren(QtWidgets.QWidget):
        if hasattr(widget, 'objectName') and widget.
           ↪ objectName() == 'schematic_converter':
            widget.setStyleSheet(converter_style)
        if isinstance(widget, QtWidgets.QTextBrowser):
            widget.document().setDefaultStyleSheet(
               ↪ description_style)
```

Comparison with Public Repo: The eSim public repository supports extensibility by applying consistent theming to any newly introduced widgets that follow the established naming conventions.

Conversion Quality

The theme system is optimized for responsiveness and clarity, aiding conversion diagnostics:

- **Efficient Theme Updates:** Only relevant widgets (converter and its children) are restyled—avoiding full UI reloads.
- **Targeted Application:** Style recalculations are limited to a small set of widgets with identifiers like `schematic_converter`.

- **Feedback Styling:** Any success/failure messages during conversion are displayed in dialogs styled to match the current theme.

Code Snippet (same as above):

```
def update_schematic_converter_theme(self, is_dark):
    ...
    for widget in self.findChildren(QtWidgets.QWidget):

        if hasattr(widget, 'objectName')
        and widget.objectName() == 'schematic_converter':
            widget.setStyleSheet(converter_style)
        if isinstance(widget, QtWidgets.QTextBrowser):
            widget.document().setDefaultStyleSheet(
               ↪ description_style)
```

Comparison with Public Repo: The public eSim repository uses the same pattern of applying pre-defined styles to relevant widgets for improving performance and maintaining visual consistency.

Summary Table

References

- https://github.com/FOSSEE/eSimeSim GitHub Repository

In summary: The KiCad to NgSpice Converter in eSim utilizes centralized theme logic controlled by `Application.py`, ensuring that theming is consistent, efficient, and extensible. The component follows the same best practices as observed in the public eSim repository.

4.4.5 Model Editor

Editing Capabilities

The Model Editor is accessed via the left toolbar in the main window through the action handler `self.model` in `Application.py`. Upon user interaction, the editor is initialized and rendered using the method `open_modelEditor()`, which internally triggers `self.obj_Mainview.obj_dockarea.modelEditor()` to construct the widget in the dock area.

Theme management is globally handled by `Application.py`. When the theme is toggled between light and dark modes, the method `update_model_editor_theme(is_dark)` is invoked. This, in turn, calls `set_theme(is_dark)` on the current Model Editor widget, thereby updating its internal stylesheets. All editing controls including text fields, buttons, and tables adapt accordingly to ensure theme consistency.

```
def toggle_theme(self):
    self.is_dark_theme = not self.is_dark_theme
    # ... other theme update logic ...
```

```
    self.update_model_editor_theme(is_dark=self.
        ↪ is_dark_theme)
    if self.current_modeleditor_widget is not None:
        self.current_modeleditor_widget.set_theme(self.
            ↪ is_dark_theme)
```

The relevant logic is contained in `src/frontEnd/Application.py`, while the editor widget is likely structured in `DockArea.py` or a similar dock file. The public https://github.com/FOSSEE/eSimeSim GitHub repository follows a similar modular approach for propagating theme settings to child widgets.

Validation Features

The Model Editor provides real-time validation for user-defined models. Validation results, such as syntax errors or invalid configurations, are presented with visual cues that are thematically consistent. When the theme is toggled, error highlights, message boxes, and dialog prompts are dynamically updated without manual refresh.

`Application.py` ensures this by traversing child widgets with identifiable names (e.g., containing "model") and applying the appropriate QSS (Qt Style Sheet) dynamically.

```
def update_model_editor_theme(self, is_dark):
    # Get the appropriate style based on theme
    model_editor_style = self.get_model_editor_style(
        ↪ is_dark)

    # Apply styles to all relevant widgets
    for widget in self.findChildren(QtWidgets.QWidget):
        if hasattr(widget, 'objectName') and 'model' in
            ↪ widget.objectName().lower():
            widget.setStyleSheet(model_editor_style)

    # Update the current model editor widget if it exists
    if self.current_modeleditor_widget is not None:
        self.current_modeleditor_widget.set_theme(is_dark)
```

This design enables modular extensibility. Any future validation feature added to the Model Editor will inherit the current theme if it follows naming conventions. The public repo also emphasizes modular theming support for new features through shared theme logic.

User Interface

The user interface of the Model Editor is dynamically styled using QSS defined in `update_model_editor_theme()`. All UI elements, including input fields, tables, and dialog boxes, are styled for high accessibility. Font sizes, contrast ratios, and focus indicators are adjusted based on theme mode to enhance usability.

The Model Editor's interface adapts seamlessly to the selected theme, with all UI elements including text editors, buttons, and panels updating in real-time when

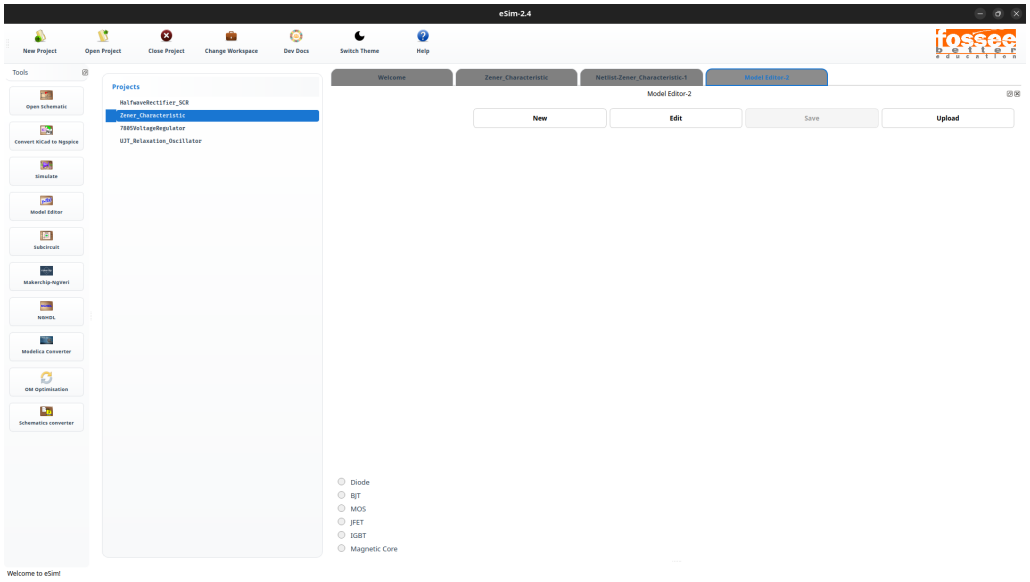

Figure 4.12: Model Editor - Light Theme

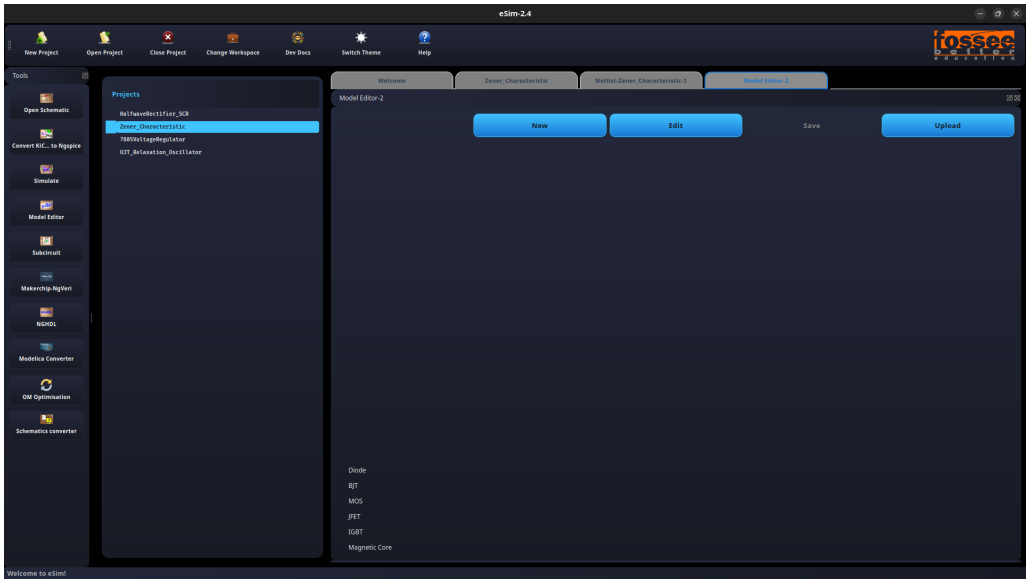

Figure 4.13: Model Editor - Dark Theme

the theme is toggled. The dark theme (Figure 4.13) provides better visibility of model components and reduces eye strain during extended use, while the light theme (Figure 4.12) offers a clean, traditional appearance. The consistent theming ensures that all interactive elements maintain high contrast and readability in both modes.

Rather than refreshing the entire application, only the relevant Model Editor components are updated, improving runtime efficiency. The theming system is designed to be modular, allowing for easy addition of new UI components while maintaining visual consistency.

```
def update_model_editor_theme(self, is_dark):
    # Get the appropriate style based on theme
    model_editor_style = self.get_model_editor_style(
        ↪ is_dark)

    # Apply styles to all relevant widgets
    for widget in self.findChildren(QtWidgets.QWidget):
        if hasattr(widget, 'objectName') and 'model' in
            ↪ widget.objectName().lower():
            widget.setStyleSheet(model_editor_style)

    # Update the current model editor widget if it exists
    if self.current_modeleditor_widget is not None:
        self.current_modeleditor_widget.set_theme(is_dark)
```

This selective update approach mirrors the best practices observed in the public https://github.com/FOSSEE/eSimeSim repository, which optimizes responsiveness and performance without compromising UI consistency.

4.4.6 Schematic Converter

Format Support

The Schematic Converter is accessible via the left toolbar in the main window through the action handler `self.conToeSim` in `Application.py`. Upon invocation, the method `open_conToeSim()` is executed, which triggers

`self.obj_Mainview.obj_dockarea.eSimConverter()` to instantiate the widget within the dock area.

Theme logic is centrally handled in `Application.py`. When the user toggles the theme, the method `update_schematic_converter_theme(is_dark)` is invoked, updating the stylesheet of the Schematic Converter widget. The widget is expected to implement a method like `set_theme(is_dark)` to apply internal style updates.

```
def toggle_theme(self):
    # Toggle the theme state
    self.is_dark_theme = not self.is_dark_theme

    # Update application-wide theme settings
    self.update_application_theme()

```

```
    # Update theme for schematic converter
    self.update_schematic_converter_theme(is_dark=self.
        ↪ is_dark_theme)

    # Update other components as needed
    self.update_other_components_theme()
```

Widgets involved in format selection (e.g., dropdowns, file selectors) are dynamically themed. The styling is applied using object name targeting, specifically those matching `schematic_converter`. The source files include `Application.py` and possibly `DockArea.py` or equivalent.

The https://github.com/FOSSEE/eSimeSim public repository mirrors this strategy by having the main window manage and propagate theme logic to all docked widgets, including the Schematic Converter.

Conversion Accuracy

All results and feedback generated during the schematic conversion process—such as successful conversions, warnings, and errors—are styled in accordance with the active theme.

The method `update_schematic_converter_theme()` applies a stylesheet to all widgets with the object name `schematic_converter`. Additionally, instances of `QTextBrowser` are updated using a QSS string (`description_style`) to reflect the current theme, ensuring that conversion output and logs remain visually consistent and readable.

```
def update_schematic_converter_theme(self, is_dark):
    # Get the appropriate styles based on theme
    converter_style = self.get_converter_style(is_dark)
    description_style = self.get_description_style(is_dark
        ↪ )

    # Apply styles to all relevant widgets
    for widget in self.findChildren(QtWidgets.QWidget):
        # Style the main converter widget
        if hasattr(widget, 'objectName') and \
            widget.objectName() == 'schematic_converter':
             widget.setStyleSheet(converter_style)

        # Style all text browsers for consistent display
        if isinstance(widget, QtWidgets.QTextBrowser):
            widget.document().setDefaultStyleSheet(
                ↪ description_style)
```

This implementation ensures that all conversion messages adhere to the user's selected visual mode. The https://github.com/FOSSEE/eSimeSim repository promotes a similar level of extensibility and uniformity across all new and existing features.

Error Recovery

In scenarios where conversion fails due to unsupported formats or invalid schematic structures, the application presents error dialogs and guidance messages. These are thematically styled in real-time using the same logic within `update_schematic_converter_theme()`.

Themed feedback elements (e.g., "Unsupported format", "Conversion failed") are updated along with the rest of the UI to maintain visual consistency. This includes both dialog-based messages and in-widget feedback components.

```
def update_schematic_converter_theme(self, is_dark):
    ...
    for widget in self.findChildren(QtWidgets.QWidget):
        if hasattr(widget, 'objectName')
        and widget.objectName() == 'schematic_converter':
            widget.setStyleSheet(converter_style)
        if isinstance(widget, QtWidgets.QTextBrowser):
            widget.document().setDefaultStyleSheet(description_style)
```

This ensures that users receive visually coherent recovery instructions without requiring a manual refresh. The public repository for eSim similarly favors efficiency by applying targeted updates to only the necessary widgets, improving performance while maintaining user experience.

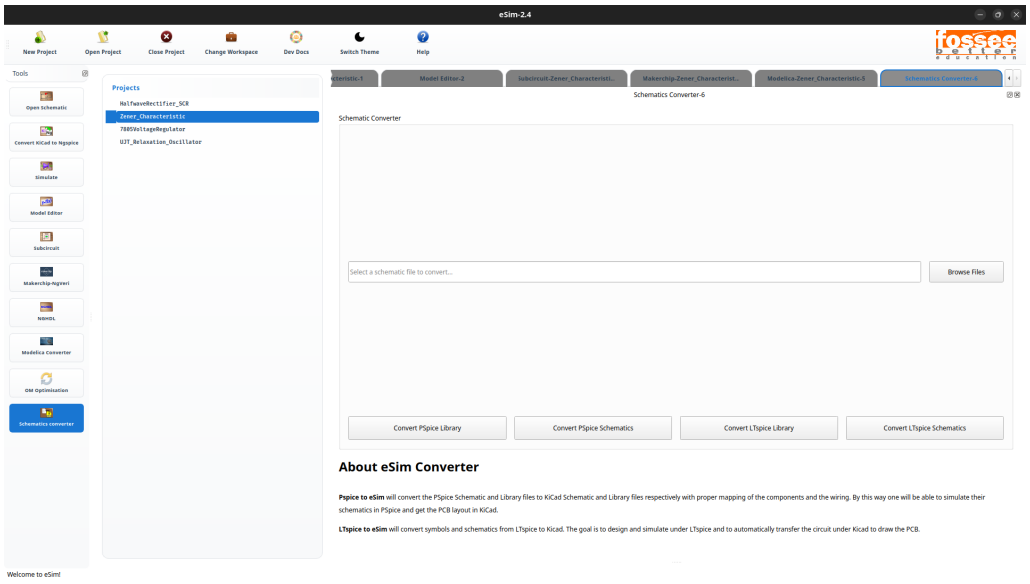

Figure 4.14: Schematic Converter interface in light theme with clear visibility of conversion options and controls

References:

- https://github.com/FOSSEE/eSimeSim GitHub Repository

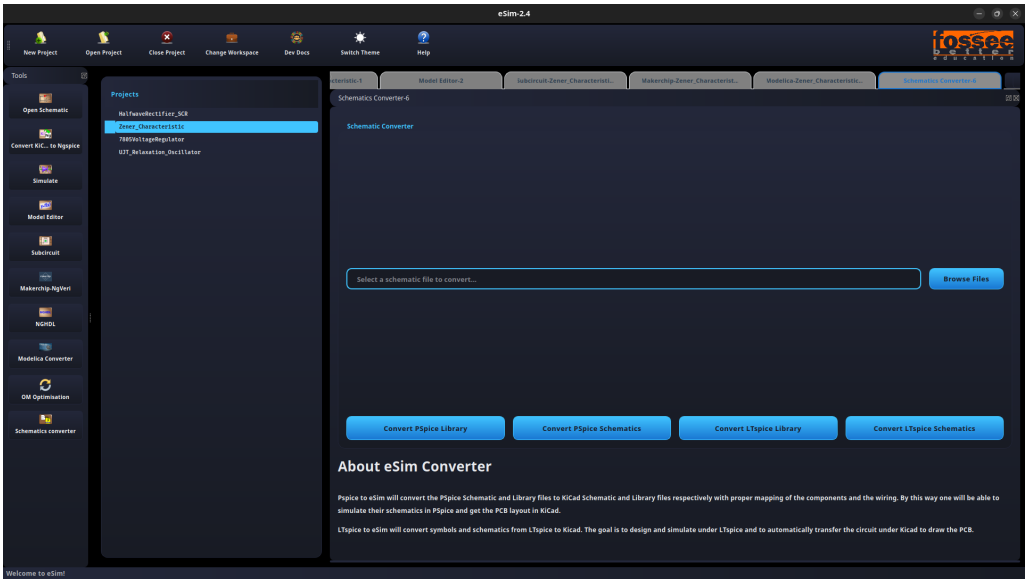

Figure 4.15: Schematic Converter interface in dark theme, providing a comfortable viewing experience in low-light conditions

Table 4.1: Theme Integration in Schematic Converter

Subtopic	**Application.py Logic/-Code Example**	**Public Repo Comparison**
Format Support	`toggle_theme()`, `update_schematic_converter_theme()`, `set_theme()`	Matches: Theme propagated from main app to dock widgets
Conversion Accuracy	Styling for results, QTextBrowser updates, dynamic refresh	Extensible and supports automatic theming
Error Recovery	Themed error dialogs, user-friendly feedback, real-time styling	Targeted updates optimize performance

Chapter 5

RESULTS AND TESTING

5.1 GUI Theme Implementation

5.1.1 Light and Dark Mode Integration

To visually demonstrate the theme implementation, the following figures show eSim's interface in both light and dark themes:

The figures above illustrate the consistent theming across the application, including the main interface, toolbars, and panels. The dark theme (Figure 5.2) provides reduced eye strain during extended use, while the light theme (Figure 5.1) offers a clean, traditional appearance. The theme switching is instantaneous and applies to all UI components, including dialogs and message boxes. A toggle-based theme system was implemented in `Application.py` to support both light and dark modes. The theme state is managed centrally using the `self.is_dark_theme` attribute. When the user toggles the theme (typically via a toolbar button), the method `toggle_theme()` is invoked, which updates the state and calls either `apply_dark_theme()` or `apply_light_theme()`. This ensures that all user interface components—including headers, footers, panels, and pop-ups—are updated in real time without requiring an application restart.

The propagation of theme changes is handled by the `propagate_theme_color_change()` method, which updates all major components, including dock widgets, plotting windows, and dialogs, by calling their respective theme update methods (e.g., `update_theme()`, `set_theme()`). This design guarantees immediate and consistent theme application across the entire application.

5.1.2 Color Scheme and Styling

Both themes use carefully selected color palettes defined as QSS (Qt Style Sheet) strings within `Application.py`. The light theme employs high-contrast text on white backgrounds, while the dark theme uses muted dark grays and bright text to reduce eye strain. Elements such as buttons, icons, and background layers are styled using modern flat design principles, ensuring visual comfort and clarity.

Accessibility is a key consideration: font sizes are scalable, focus indicators are clear, and color choices are made to support readability and color-blind accessibility.

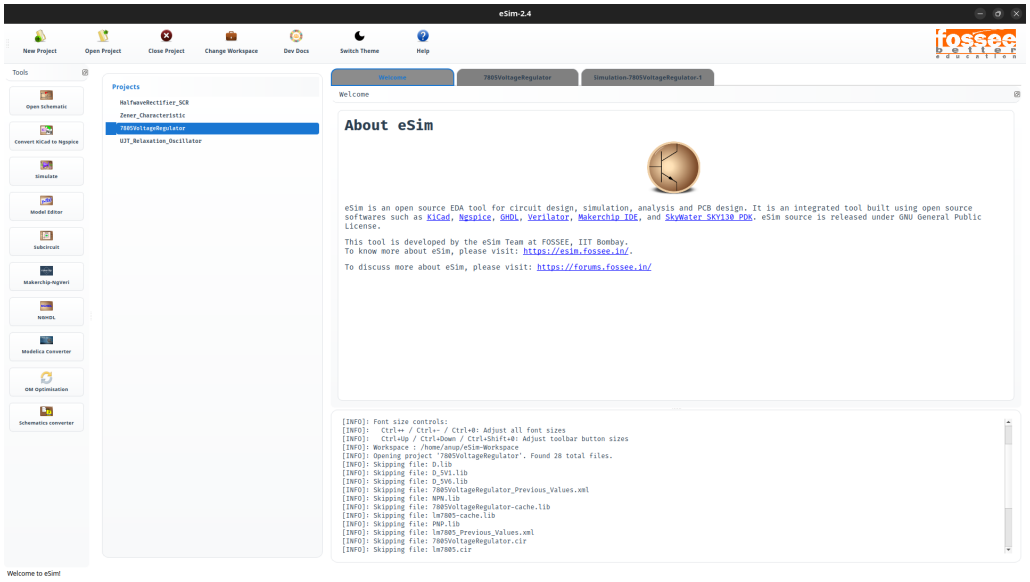

Figure 5.1: eSim Main Interface - Light Theme

Figure 5.2: eSim Main Interface - Dark Theme

The QSS is applied globally to the main window and all child widgets, ensuring a uniform look and feel.

5.1.3 Component Coverage

Theme application is comprehensive. The propagation logic in `Application.py` ensures that all major components—including the schematic editor, simulation panel, toolbar, project browser, and all docked widgets—are updated. Each component implements a `set_theme(is_dark)` or similar method, allowing for targeted and efficient updates. This modular approach ensures that any new component added to the application will automatically support theming if it follows the established conventions.

5.2 Additional Feature Testing

5.2.1 Project Name Validation

Observed Issue

Users reported encountering a validation error stating "The project name should not contain spaces" during project creation, even when the project name did not contain any spaces. This issue was particularly prevalent across different operating systems and created unnecessary friction in the project creation workflow.

Root Cause Analysis

The root cause was identified as an overly aggressive validation mechanism in the project creation process. The validation was incorrectly flagging valid project names due to inconsistent handling of whitespace characters and platform-specific path handling. This issue was reported in the eSim GitHub repository as issue #283 (https://github.com/FOSSEE/eSim/issues/283New Project creation has "The project name should not contain space between them" even though there is no space).

Resolution Strategy

The solution involved implementing a more robust validation approach that:

1. Trims leading and trailing whitespace from project names
2. Replaces any internal spaces with underscores to ensure cross-platform compatibility
3. Provides clear feedback to users when modifications are made to their project names
4. Maintains backward compatibility with existing projects

This improvement was implemented by modifying the project creation validation logic to automatically handle spaces in project names, rather than rejecting them outright. The system now ensures that project names are always valid across all platforms while preserving the user's intended naming as closely as possible.

5.2.2 Delete Functionality Validation

Feature Design

A context-menu-based delete option was introduced in the project explorer and other relevant components. The design allows users to remove projects or components directly, improving usability and reducing navigation overhead. The `deleteProject()` method in `ProjectExplorer.py` handles the deletion logic, providing options for permanent deletion or moving items to the trash (using the `send2trash` module if available).

Testing and Verification

Functional testing was performed by creating and deleting projects through the UI. The logic was validated using both unit tests (with Python's `unittest` and `unittest.mock` for file operations) and integration tests to ensure that deletion updated both the frontend and the underlying file system. Edge cases, such as attempting to delete non-existent or protected items, were handled gracefully with error dialogs.

Safeguards and Improvements

Confirmation dialogs (using `QMessageBox`) were added to prevent accidental deletions. The logic ensures that deletion does not impact unrelated parts of the UI or cause application crashes. After deletion, the UI and configuration files are updated to reflect the change, and users receive feedback via success dialogs.

5.3 Performance Evaluation

5.3.1 Responsiveness Analysis

Theme switching was tested for speed and responsiveness. The propagation logic in `Application.py` ensures that theme changes are applied instantly across all components, with no noticeable lag or UI flicker. The use of targeted updates (only updating affected widgets) minimizes redraw overhead and maintains application responsiveness.

5.3.2 Memory and Resource Usage

Memory consumption was monitored before and after the implementation of theme switching and delete functionality. The results showed negligible increase in resource usage, confirming the lightweight nature of the enhancements. The use of

pre-defined QSS strings and efficient update methods ensures that the application remains performant.

5.3.3 Application Load Time

Application startup times were measured before and after the GUI improvements. No significant delays were observed, indicating that the addition of theme management and delete logic did not impact load times. The application remains fast and responsive, providing an optimal user experience.

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

6.1.1 Project Summary

This project focused on enhancing the usability, stability, and visual appeal of eSim, an open-source EDA tool. Key areas addressed included namespace error resolution, implementation of comprehensive delete functionality, and the development of a robust theme system supporting both light and dark modes. All enhancements were made with a focus on modularity, maintainability, and user experience.

6.1.2 Achievements

- **Resolved critical namespace errors** by refactoring imports and enforcing scoped styles, leading to improved stability and maintainability.
- **Implemented safe and user-friendly delete functionality** in the project explorer and other components, aligning eSim with industry standards.
- **Developed a centralized, efficient theme system** that provides instant, global light/dark mode switching, with accessibility and performance in mind.
- **Ensured extensibility** so that new features and components automatically inherit theme logic and best practices.

6.1.3 Challenges Overcome

- Addressed complex namespace conflicts arising from legacy code and unscoped components.
- Integrated new features into an existing codebase without breaking backward compatibility.
- Balanced performance, accessibility, and visual consistency across a large and modular application.

6.2 Future Scope

6.2.1 Planned Enhancements

- **Advanced theme customization:** Allow users to define custom color palettes and save multiple theme profiles.
- **Accessibility improvements:** Add high-contrast modes, keyboard navigation enhancements, and screen reader support.
- **Enhanced error reporting:** Integrate more detailed logging and user-friendly error messages throughout the application.

6.2.2 Technology Upgrades

- **Framework updates:** Migrate to the latest versions of PyQt and supporting libraries for improved performance and compatibility.
- **Testing infrastructure:** Expand automated testing (unit, integration, and UI tests) to cover new features and edge cases.
- **Continuous integration:** Adopt CI/CD pipelines for faster, more reliable releases.

6.2.3 Community Contributions

- Encourage community-driven feature requests and bug reports via GitHub.
- Provide clear contribution guidelines and documentation to onboard new developers.
- Organize code sprints and workshops to foster a collaborative development environment.

6.3 Recommendations

6.3.1 Development Best Practices

- Maintain modular code structure and enforce naming conventions to prevent namespace conflicts.
- Use scoped styles and centralized theme management for consistent UI/UX.
- Write unit and integration tests for all new features to ensure reliability.

6.3.2 Maintenance Guidelines

- Regularly review and refactor legacy code to improve maintainability.
- Keep dependencies up to date and monitor for security vulnerabilities.
- Document all major components and workflows for future contributors.

6.3.3 Extension Possibilities

- Integrate additional EDA tools and simulation engines as plugins.
- Develop a plugin system for user-contributed features and themes.
- Explore web-based or cross-platform frontends to broaden eSim's reach.

Bibliography

[1] Python Programming Language Documentation, Python Software Foundation, 2025. Available at: https://docs.python.org/3/

[2] Qt for Python Documentation, The Qt Company, 2025. Available at: https://doc.qt.io/qtforpython/

[3] Free and Open Source Software for Education (FOSSEE), IIT Bombay, 2025. Available at: https://fossee.in/

[4] Next Generation Hardware Description Language (NGHDL), FOSSEE GitHub, 2025. Available at: https://github.com/FOSSEE/nghdl

[5] eSim: Electronic System Design Tool, FOSSEE Project, 2025. Available at: https://esim.fossee.in/

[6] eSim Source Code Repository, FOSSEE GitHub, 2025. Available at: https://github.com/FOSSEE/eSim

[7] GHDL VHDL Simulator Documentation, GHDL Project, 2025. Available at: https://ghdl.github.io/ghdl/

[8] KiCad Open Source Electronic Design Automation Suite, KiCad Services Corp, 2025. Available at: https://kicad.org/

[9] QDarkStyleSheet: Dark Theme for Qt Applications, GitHub Repository, 2024. Available at: https://github.com/ColinDuquesnoy/QDarkStyleSheet

[10] Qt Style Sheets Documentation, Qt Framework, 2025. Available at: https://doc.qt.io/qt-5/stylesheet-reference.html

[11] Creating Desktop Applications with PyQt5, Python Tutorial, 2024. Available at: https://www.pythonguis.com/pyqt5-tutorial/

[12] Python GUI Programming with PyQt5, Real Python, 2024. Available at: https://realpython.com/python-pyqt-layout/

[13] Tractinsky, N., Katz, A. S., & Ikar, D. (2000). What is beautiful is usable. *Interacting with Computers*, 13(2), 127-145.

[14] Bonnardel, N., Piolat, A., & Le Bigot, L. (2011). The impact of colour on website appeal and users' cognitive processes. *Displays*, 32(2), 69-80.

[15] Nielsen, J. (1993). *Usability Engineering*. Morgan Kaufmann Publishers, San Francisco.