eSim Semester Long Internship Spring 2026

On

KiCad Design Diary: A Native Plugin for Tracking PCB Changes and eSim Simulation Checkpoints

Submitted by

Sia Upadhyay

B.Tech CSE, 3rd Year

VIT Bhopal University

Under the guidance of

Prof. Prabhu Ramachandran

Principal Investigator

Department of Aerospace Engineering

Indian Institute of Technology Bombay

May 2026

Acknowledgment

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be part of the FOSSEE internship programme. His leadership in advancing open-source engineering software has been an inspiration throughout this work.

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

My sincere appreciation goes to my mentor, Mr. Sumanto Kar, for his continuous technical guidance, constructive feedback, and encouragement at every stage of the project. His insights were crucial in shaping the plugin architecture, particularly the decision to delegate netlist generation to eSim's existing converter and the design of the rollback safety mechanism. The hash-based staleness detection described in Chapter 4 took its final form following one of these review sessions.

I would also like to thank my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K, for their coordination, technical inputs, and timely reviews during the internship.

This work gave me the opportunity to study KiCad's plugin architecture, integrate eSim's converter pipeline with NGSpice for SPICE simulation, and apply software engineering practices such as content-addressed storage, set-theoretic diffing, and pattern-based equivalence detection to a real circuit-design workflow. The experience of translating these ideas into functional software, and then writing the mathematical foundations down clearly enough for the next intern to extend them, has been deeply enriching.

I also thank the entire FOSSEE team for their support and coordination throughout the duration of this internship.

Sia Upadhyay
VIT Bhopal University
May 2026

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

Electronic Design Automation (EDA) tools are essential to hardware engineering education. In the open-source academic ecosystem, KiCad is widely used for PCB layout and schematic capture, while eSim (developed by FOSSEE, IIT Bombay) provides SPICE-based simulation through NGSpice as the backend engine.

Designing a printed circuit board is rarely completed in a single sitting. Components are reassigned, tracks rerouted, parts swapped for ones that are available, and these revisions add up over days or weeks. The history of those decisions is itself part of the design, but KiCad does not keep it. The undo history is volatile and is discarded when the project closes. The file system shows only the current state. Asking "what changed since last Tuesday?" has no answer the tool can give.

In software, this kind of history comes for free. Git has been the default for over a decade [8]: every change can be traced back to an author, a time, and a message. Hardware design has no such default. KiCad stores its files as plain-text S-expressions [6], which means they *could* be versioned with Git, and some advanced users do exactly that. But for the typical eSim audience (ECE students, researchers, and educators) Git is a non-trivial barrier. Most users either copy and rename files (`schematic_v1`, `schematic_v2_final`), or rely on memory. The cost shows up later, when a board that worked previously stops working and there is no log of what changed in between.

This internship addresses that gap through the development of a KiCad plugin that captures design history automatically and integrates simulation results into the same record.

1.2 Overview of eSim

eSim is a free and open-source EDA tool developed by FOSSEE, IIT Bombay [3], distributed under the GNU General Public Licence. It is not a single tool but an integrated stack: KiCad handles schematic capture and PCB layout, NGSpice [4] handles analog and mixed-signal SPICE simulation, NGHDL and GHDL handle VHDL, and Verilator and Makerchip handle Verilog. eSim's role is to integrate these tools and, in particular, to translate between their file formats.

The component most directly relevant to this project is the KiCad-to-NGSpice converter, which produces a SPICE netlist from a KiCad schematic. KiCad itself exposes a Python API, specifically the `pcbnew` module for the PCB editor and direct access to the schematic file format [2], which is what makes plugins such as Design Diary possible. eSim handles simulation, KiCad handles design, and the plugin sits one layer above both, tracking how a design evolves over time and binding each simulation result back to the exact schematic state that produced it.

1.3 Objectives of the Project

The primary objectives of this project are listed below.

1. Develop a KiCad plugin that captures schematic and PCB snapshots automatically on every save, with no manual action required.
2. Implement a semantic diff engine that reports changes between consecutive snapshots in plain English rather than as raw text differences.
3. Provide an in-KiCad wxPython timeline panel with rollback to any earlier snapshot.
4. Integrate the eSim/NGSpice pipeline so that each simulation run becomes a first-class entry in the diary, with its waveform tied to the schematic state that produced it.
5. Detect when a recorded simulation no longer reflects the current schematic and flag it as stale, using a hash-based staleness mechanism.
6. Detect circuits that NGSpice cannot simulate (microcontrollers, digital logic boards, connector-heavy designs) and warn the user rather than producing cryptic SPICE errors.
7. Export the full history, with embedded interactive waveforms, as a single self-contained HTML file.
8. Validate the result on three real circuits exercising three different code paths: transient analysis, AC analysis, and non-simulatable detection.

1.4 Novel Contributions

This project makes the following contributions that are not present in prior work on the eSim and KiCad ecosystem.

- **Automatic semantic snapshot capture.** The plugin captures both schematic and PCB state on every save, with zero user configuration, and reports changes in human-readable form rather than as S-expression-level diffs.

- **Hash-based stale simulation detection.** A SHA-256 hash of the schematic recorded at simulation time is compared against the current schematic on every timeline render, automatically flagging simulations that no longer reflect the current design.
- **Pattern-based non-simulatable circuit detection.** A regular expression pattern table identifies microcontrollers, EEPROMs, digital logic, and connector-heavy boards before NGSpice is invoked, replacing cryptic SPICE errors with a clear plain-language explanation.
- **Reversible rollback.** An automatic pre-rollback snapshot makes the rollback operation itself reversible, so that no information is lost on misclicks.
- **Self-contained interactive HTML export.** A single HTML file with all CSS, JavaScript, and waveform data inlined provides a portable record that can be emailed or attached to a project submission without broken links.

1.5 Methodology Overview

The project was executed in five phases. Phase 1 covered study of the KiCad Action-Plugin API and the S-expression schematic format. Phase 2 involved implementing snapshot capture and the semantic diff engine. Phase 3 covered the wxPython UI, rollback mechanism, and component history view. Phase 4 dealt with eSim and NGSpice integration, including hash-based staleness and non-simulatable detection. Phase 5 was end-to-end testing on three real circuits and report writing. The full source code is hosted on GitHub [11].

1.6 Organisation of the Report

The remainder of this report is structured as follows. Chapter 2 surveys the technologies and prior work that the plugin builds on. Chapter 3 formalises the problem statement and the proposed solution. Chapter 4 presents the mathematical foundations of the plugin: the set-theoretic formulation of the diff engine, the hash-based staleness model, the regular-language formulation of non-simulatable detection, and complexity analysis of the principal operations. Readers who care only about the end-to-end behaviour may skim this chapter; readers who intend to extend the plugin should read it in full. Chapter 5 describes the implementation of the core plugin (snapshot capture, diffing, timeline UI, rollback, HTML export). Chapter 6 covers the simulation pipeline (eSim converter, NGSpice batch invocation, stale detection, non-simulatable detection, error translation). Chapter 7 presents end-to-end validation on three test circuits. Chapter 8 concludes and outlines limitations and future work. The daily work log is included as Appendix A.

Chapter 2

Literature Survey

2.1 KiCad and the S-Expression File Format

KiCad stores both schematic files (`.kicad_sch`) and PCB files (`.kicad_pcb`) as plain-text S-expressions, a Lisp-derived nested parenthetical syntax [6]. Because the file format is text-based, line-level differences can in principle be tracked by any general purpose version control system. In practice, the syntactic detail of these files (parentheses, internal identifiers, automatically-generated UUIDs) makes line-level diffs difficult to read, which is one of the motivations for the semantic diff engine described in Chapters 4 and 5.

The schematic file is a top-level S-expression whose nested children encode component symbols, library identifiers, reference designators, values, footprints, net labels, and graphical positions. The PCB file follows the same convention for footprints, copper layers, vias, and zones. Both files are designed to be readable to humans, but they are not designed to be *diffable* in a human-meaningful way; a single drag of a component in the GUI may produce several syntactic changes scattered across the file.

2.2 KiCad Python Scripting API

KiCad version 9 exposes a Python module called `pcbnew` [2] that allows developers to register `ActionPlugin` subclasses and access footprints, pad nets, component values, and board geometry programmatically. This API is the entry point for the plugin and enables automatic capture of the live PCB state on every save event. The schematic file format, although not exposed through a stable Python API at the symbol level, is parsed directly as S-expressions by the plugin.

The `ActionPlugin` mechanism follows a simple registration pattern: a subclass declares its metadata in a `defaults()` method and implements the user-facing entry point in a `Run()` method. KiCad scans its plugin directory at startup, imports each module, and registers any plugins it finds. This pattern means the plugin has no fixed installation step beyond placing a folder in the correct location.

2.3 SPICE and NGSpice

SPICE was developed at UC Berkeley in 1973 and remains the standard for analog circuit simulation. NGSpice [4] implements the SPICE3 model set and is the simulation backend used by eSim. Its batch mode (the `-b` flag) is used by the plugin for headless simulation, meaning NGSpice reads a netlist, executes the specified analysis, writes results to a raw file, and exits. No interactive prompt, plot windows, or terminal emulator is involved, which makes batch mode suitable for invocation from within another application.

NGSpice supports several analysis types relevant to this work. Transient analysis (`.tran`) integrates the differential equations of the circuit forward in time and produces a time-domain waveform. AC analysis (`.ac`) linearises the circuit around its DC operating point and sweeps a small-signal source through a frequency range. DC operating point (`.op`) solves the circuit's algebraic equations at zero frequency. The plugin records the analysis type as part of each checkpoint so that the HTML report can present each waveform with the correct axis labels.

2.4 eSim and the KiCad-to-NGSpice Converter

eSim [3] is the open-source EDA tool developed by FOSSEE, IIT Bombay. The most directly relevant component for this project is its KiCad-to-NGSpice converter, which translates a KiCad schematic into a SPICE netlist suitable for NGSpice batch invocation. The plugin invokes this converter directly rather than reimplementing netlist generation, both to avoid duplicating eSim's existing work and to ensure that the plugin remains compatible with future eSim changes. An earlier FOSSEE intern arrived at the same decision for the same reason [9], and reading that report confirmed the approach.

2.5 Version Control and Hardware Design

Git [8] has been the dominant version control system for software development for over a decade. Several projects have explored applying Git to KiCad designs, with varying degrees of success. The fundamental limitation is the cognitive load: commits, staging, branches, and merges all have to be understood, and meaningful commit messages have to be written manually on every change. Even when Git is used, its diff is a text-level diff of S-expressions, which is opaque to someone who reads diffs only occasionally. Community KiCad diff scripts have shown that semantic diffing is feasible, but they are mostly command-line tools that do not integrate with KiCad's UI or with eSim's simulation pipeline.

2.6 Hash-Based Content Addressing

The use of cryptographic hashes to detect content changes is a well-established technique in software systems. Git itself uses SHA-1 (and more recently SHA-256) to identify content by its hash rather than by location. The plugin applies the

same principle at a smaller scale: every recorded simulation stores a SHA-256 hash of the schematic at the time of simulation, and any later change to the schematic that produces a different hash automatically marks the recorded simulation as stale. The hash function is applied after whitespace normalisation per line, to tolerate line-ending differences between Windows (CRLF) and Linux (LF) on the same project. The full mathematical treatment of this mechanism appears in Section 4.3.

2.7 Regular Languages and Pattern-Based Classification

The non-simulatable detection mechanism is, in formal terms, a recogniser for a finite union of regular languages: each pattern in the table defines a regular language, and a board is classified as non-simulatable if any of its component reference designators or values falls in the union. This view is elaborated in Section 4.4. The same view explains why the pattern table can be extended without touching any other code: a new pattern simply adds another regular language to the union.

2.8 Existing Tools and Limitations

KiCad's built-in undo history is volatile and is discarded when the project closes. The file system shows only the current state. eSim's simulation pipeline operates per-invocation and does not record a history of past runs. No existing open-source tool integrates snapshot history, semantic diffing, and simulation checkpoints in a single zero-configuration plugin. This work fills that gap.

Chapter 3

Problem Statement

3.1 Problem Statement

A KiCad user working on a long-lived PCB design faces the following challenges.

1. **No persistent change history.** KiCad's undo stack is volatile and is discarded on project close. There is no record of what changed between sessions.
2. **No semantic diff between revisions.** Even when files are kept under version control, the resulting diffs are at the level of S-expression syntax, which is difficult to read and does not surface the engineering content of the change.
3. **Disconnected simulation history.** Each NGSpice run is a separate event with no record of the schematic state that produced it. When a simulation result no longer matches the current schematic, there is no warning.
4. **Manual file management.** Users typically resort to renaming files (`board_v1.kicad_pcb`, `board_v2_final.kicad_pcb`) to keep historical states, which provides only the contents and not the changes between them.
5. **Unhelpful errors on non-simulatable circuits.** Invoking NGSpice on a board built around microcontrollers, EEPROMs, or connector arrays produces a wall of cryptic SPICE errors that are difficult for non-expert users to interpret.

3.2 Approach

The proposed solution is a KiCad ActionPlugin that performs the following operations automatically.

- Automatic capture of schematic and PCB snapshots on every save event, via the `pcbnew` ActionPlugin mechanism.
- Semantic diff between consecutive snapshots, reporting changes as plain-English strings keyed by component reference.
- In-KiCad wxPython timeline panel with colour-coded rows by event type (schematic edit, PCB edit, simulation, rollback).

- Rollback to any earlier snapshot, with an automatic pre-rollback safety snapshot.
- Per-component history view, replaying all changes affecting a chosen reference designator.
- NGSpice batch-mode simulation through eSim's existing converter, with each run recorded as a checkpoint bound to the schematic hash.
- SHA-256 hash-based staleness detection, automatically flagging checkpoints whose schematic has since changed.
- Pattern-based non-simulatable circuit detection, replacing NGSpice errors with a plain-language explanation.
- Single-file HTML export with embedded interactive waveform plots, summary statistics, and the full change timeline.

The plugin is non-intrusive. It does not modify any KiCad project files during normal operation (other than the rollback action, which is explicitly user-initiated), and operates purely as an observer and analysis tool.

Chapter 4

Mathematical Foundations

This chapter sets out the mathematical structures the plugin uses internally, so that a future intern can reason about the behaviour of each component without reading the source. The plugin is deliberately built on small, plain mathematical objects: finite sets, mappings between them, a cryptographic hash function, a finite union of regular languages, and a few asymptotic bounds. Nothing here is unusual; the value of writing it down is that the implementation can then be checked against the specification.

4.1 The Snapshot as a Mapping

Let $\mathcal{R}$ denote the set of all reference designators used in a KiCad project (for example $\{R_1, R_2, U_1, C_1, \ldots\}$) and let $\mathcal{P}$ denote the set of component property tuples, where each tuple records the component's value, footprint, and PCB position. A *snapshot* captured at time t is a partial function

$$S_t : \mathcal{R} \rightharpoonup \mathcal{P}, \qquad S_t(r) = (v_r, f_r, x_r, y_r),$$

defined exactly on those references that are present in the design at time t. The notation $\mathrm{dom}(S_t)$ refers to the set of references on which S_t is defined, and $|S_t|$ refers to the cardinality of this set, i.e. the number of components in the snapshot.

Snapshots are stored on disk as JSON files keyed by reference designator. The flat structure of the mapping (one component per key, properties as sub-fields) is preserved in the JSON layout so that diffing reduces to set and dictionary operations rather than tree traversal.

4.2 The Semantic Diff Engine

Given two consecutive snapshots S_t and S_{t+1}, the diff engine partitions the symmetric difference of their domains into the sets of added and removed references, and identifies modified references by comparing properties on the intersection. Formally,

define

$$\text{Added}(S_t, S_{t+1}) = \text{dom}(S_{t+1}) \setminus \text{dom}(S_t), \tag{4.1}$$

$$\text{Removed}(S_t, S_{t+1}) = \text{dom}(S_t) \setminus \text{dom}(S_{t+1}), \tag{4.2}$$

$$\text{Modified}(S_t, S_{t+1}) = \big\{\, r \in \text{dom}(S_t) \cap \text{dom}(S_{t+1}) \;\big|\; S_t(r) \neq S_{t+1}(r) \,\big\}. \tag{4.3}$$

The full diff is the disjoint union

$$\Delta(S_t, S_{t+1}) = \text{Added}(S_t, S_{t+1}) \;\sqcup\; \text{Removed}(S_t, S_{t+1}) \;\sqcup\; \text{Modified}(S_t, S_{t+1}).$$

For each reference in Modified, the engine further compares the component property tuples coordinate-wise and emits one change record per differing coordinate. If $S_t(r) = (v_r, f_r, x_r, y_r)$ and $S_{t+1}(r) = (v'_r, f'_r, x'_r, y'_r)$, the change record set for r is

$$\mathcal{C}(r) = \{\text{ “value” } \mid v_r \neq v'_r \,\} \cup \{\text{ “footprint” } \mid f_r \neq f'_r \,\} \cup \{\text{ “position” } \mid (x_r, y_r) \neq (x'_r, y'_r) \,\}.$$

Each element of Δ is then rendered as a plain-English string through a fixed template, giving the human-readable output that appears in the timeline panel. The algorithm is summarised below.

Algorithm 4.1 Semantic diff between two snapshots.

Require: Snapshots S_t and S_{t+1}, each a mapping $\mathcal{R} \rightharpoonup \mathcal{P}$.
Ensure: Ordered list of plain-English change strings L.

$L \leftarrow [\,]$
$A \leftarrow \text{dom}(S_{t+1}) \setminus \text{dom}(S_t)$
$R \leftarrow \text{dom}(S_t) \setminus \text{dom}(S_{t+1})$
$M \leftarrow \text{dom}(S_t) \cap \text{dom}(S_{t+1})$
for all $r \in A$ in sorted order **do**
    Append “*Added* r *(value* v_r*)*” to L
end for
for all $r \in R$ in sorted order **do**
    Append “*Removed* r” to L
end for
for all $r \in M$ in sorted order **do**
    if $v_r \neq v'_r$ **then**
        Append “*Changed* r *value:* $v_r \to v'_r$” to L
    end if
    if $f_r \neq f'_r$ **then**
        Append “*Changed* r *footprint:* $f_r \to f'_r$” to L
    end if
    if $(x_r, y_r) \neq (x'_r, y'_r)$ **then**
        Append “*Moved* r *to* (x'_r, y'_r)” to L
    end if
end for
return L

Determinism and purity. Δ depends only on S_t and S_{t+1}. It has no side effects, no randomness, and no global state. As a consequence, the same pair of snapshots always produces the same diff, which is a useful property for both testing and reproducibility. Re-rendering the timeline does not change any historical entry.

Complexity. Let $n = |S_t|$ and $m = |S_{t+1}|$. Each set operation in Algorithm 4.1 runs in time $O(n+m)$ over the average-case hash behaviour of Python's `set` and `dict` types, and the property comparison on the intersection adds another $O(\min(n, m))$. The total complexity is therefore $O(n + m)$, which means the cost of diffing grows linearly with the board size. In practice, a typical educational board has $n, m < 100$, so this cost is negligible compared to the cost of writing the snapshot to disk.

4.3 Hash-Based Stale Detection

Let $\sigma \in \{0,1\}^*$ denote the bytes of the schematic file (`.kicad_sch`) as read from disk, and let $H : \{0,1\}^* \to \{0,1\}^{256}$ denote the SHA-256 cryptographic hash function. The naive fingerprint of the schematic would simply be $H(\sigma)$, but this would be fragile: a one-byte change in line endings between Windows (CRLF) and Linux (LF) would produce a different fingerprint even when the engineering content is identical.

To suppress these false positives, the plugin applies a normalisation map $N : \{0,1\}^* \to \{0,1\}^*$ defined line-wise as follows. If the schematic is partitioned into lines $\sigma = \ell_1 \ell_2 \cdots \ell_k$ (splitting on LF, with CR characters discarded), then

$$N(\sigma) = N(\ell_1)\,\mathrm{LF}\,N(\ell_2)\,\mathrm{LF}\,\cdots\,N(\ell_k)\,\mathrm{LF}, \quad N(\ell) = \mathrm{rstrip}(\ell),$$

where rstrip removes trailing whitespace from a line. Leading indentation is preserved, since changes to indentation can reflect real restructuring of the file. The schematic fingerprint is then

$$\phi(\sigma) = H(N(\sigma)).$$

Equivalence relation. The map ϕ induces an equivalence relation $\sim_\phi$ on schematic byte-strings:

$$\sigma_1 \sim_\phi \sigma_2 \iff \phi(\sigma_1) = \phi(\sigma_2).$$

Two schematics related by $\sim_\phi$ are treated by the plugin as the same underlying design. This is the relation the plugin actually cares about, because it is robust to line-ending differences that occur whenever a project is moved between Windows and Linux machines.

Stale predicate. Each simulation checkpoint stores the fingerprint of the schematic at simulation time. Write this stored fingerprint as ϕ_{sim} and the current schematic fingerprint as ϕ_{now}. A checkpoint is *stale* if and only if

$$\mathrm{Stale}(\phi_{\text{sim}}, \phi_{\text{now}}) \;\equiv\; \phi_{\text{sim}} \neq \phi_{\text{now}}.$$

The check is constant-time once the current fingerprint is computed, and the current fingerprint is cached for the duration of a timeline render.

Collision probability. SHA-256 produces 256-bit digests. The probability of two distinct schematics producing the same fingerprint by accident is bounded by the birthday bound at 2^{-128}, which is negligible in any realistic use of the plugin. A faster, non-cryptographic hash such as xxHash would suffice in principle, but SHA-256 is already available in the Python standard library at zero cost, so there is no practical reason to switch.

4.4 Non-Simulatable Circuit Detection

Let Σ be the alphabet of ASCII characters used in KiCad reference designators and component values. Let

$$\mathcal{T} = \{\, (e_i, m_i) \,\}_{i=1}^{k}$$

denote the pattern table, where each e_i is a regular expression over Σ and m_i is the human-readable message associated with that pattern (for example, "PIC microcontroller" or "connector / header"). Each e_i recognises a regular language $L(e_i) \subseteq \Sigma^*$.

The plugin defines the *non-simulatable language* as the union

$$\mathcal{L}_{\mathrm{NS}} = \bigcup_{i=1}^{k} L(e_i).$$

Being a finite union of regular languages, $\mathcal{L}_{\mathrm{NS}}$ is itself regular and is recognised by the disjunction $e_1 \mid e_2 \mid \cdots \mid e_k$. The classification predicate on a snapshot S is then

$$\mathrm{NonSim}(S) \;\equiv\; \big(\exists r \in \mathrm{dom}(S) : r \in \mathcal{L}_{\mathrm{NS}}\big) \;\vee\; \big(\exists r \in \mathrm{dom}(S) : v_r \in \mathcal{L}_{\mathrm{NS}}\big),$$

where v_r is the value field of $S(r)$. The predicate fires if any reference designator or any component value matches any pattern in the table.

Table 4.1 lists the patterns currently in the table and the human-readable message each one carries. The first column is the regular expression in standard PCRE notation; the second column is the message the user sees if the pattern matches.

Table 4.1: Non-simulatable pattern table $\mathcal{T}$ as of May 2026.

Regular expression	**Message** m_i
`\bPIC [0-9A-Z_\-]*\b`	PIC microcontroller
`\bATMEGA [0-9A-Z_\-]*\b`	ATMEGA microcontroller
`\bATTINY [0-9A-Z_\-]*\b`	ATTINY microcontroller
`\b 24C[0-9A-Z]+\b`	I^2C EEPROM
`\b 74HC[0-9]+\b`	74HC-series digital logic
`\b 74LS[0-9]+\b`	74LS-series digital logic
`^P[0-9]+$`	Connector / header

Extensibility. The pattern table is extensible without any code change beyond adding a row: adding (e_{k+1}, m_{k+1}) extends $\mathcal{L}_{\text{NS}}$ to $\mathcal{L}_{\text{NS}} \cup L(e_{k+1})$, which by the closure of regular languages under union is still regular. This is the formal justification for the comment "*extended at runtime as new cases come in*" that appears in the source.

Complexity. With k patterns of total compiled NFA size K, each r-versus-table test runs in time $O(K|r|)$ using a standard NFA simulation. The full predicate $\text{NonSim}(S)$ runs in time $O\Big(K \sum_{r \in \text{dom}(S)} (|r| + |v_r|)\Big)$, which is again linear in the size of the snapshot. The pre-flight cost is therefore negligible compared to the cost of invoking NGSpice, which justifies running the pre-flight check on every simulation request.

4.5 Warning Deduplication via Fingerprint Memoisation

Without further care, the long explanatory popup described in Chapter 6 would appear every time the user clicks *Run Simulation* on a non-simulatable board, which would be irritating. To avoid this, the plugin computes a small *circuit fingerprint* that identifies the offending components in a particular board and remembers the fingerprints for which the long popup has already been shown.

Define

$$\text{NSRefs}(S) = \big\{\, r \in \text{dom}(S) \;\big|\; r \in \mathcal{L}_{\text{NS}} \;\vee\; v_r \in \mathcal{L}_{\text{NS}} \,\big\}$$

to be the set of references in S that fired the non-simulatable predicate, and let

$$\psi(S) = H\big(\text{sort}(\text{NSRefs}(S))\big)$$

be the SHA-256 hash of the canonically-sorted, comma-joined list of those references. The plugin maintains a set

$$\mathcal{W} \subseteq \{0,1\}^{256}$$

of fingerprints already warned about, persisted as `_sim_warning_shown.json`. On each *Run Simulation* click on a non-simulatable board:

- If $\psi(S) \notin \mathcal{W}$, the long popup is shown and $\psi(S)$ is added to $\mathcal{W}$.
- If $\psi(S) \in \mathcal{W}$, only the short inline notice is shown.

The choice of $\text{NSRefs}(S)$ rather than the full $\text{dom}(S)$ means that two boards differing only in non-flagged parts share the same fingerprint and therefore share a single warning, which matches user expectations: if the offending parts are the same, the warning has already been seen.

4.6 Snapshot Ordering and Lexicographic Time

Snapshot filenames are constructed from a UTC timestamp of the form `YYYYMMDD_HHMMSS`, so that the lexicographic order on filenames agrees with the chronological order on

capture times. Formally, if $\tau(f)$ denotes the timestamp extracted from filename f, then for any two snapshot filenames f_1 and f_2,

$$f_1 <_{\text{lex}} f_2 \iff \tau(f_1) < \tau(f_2).$$

This identity is what allows the timeline panel to render snapshots in chronological order by sorting filenames as strings, without ever parsing the timestamp into a date object. The same identity makes *the most recent snapshot* equal to *the lexicographically largest filename*, which simplifies the rollback target selection.

4.7 Complexity Summary

Table 4.2 collects the worst-case asymptotic running times of the principal operations in the plugin. Here $n = |S_t|$ and $m = |S_{t+1}|$ are snapshot sizes, $|\sigma|$ is the size of the schematic in bytes, T is the number of snapshots in the history, and k is the number of patterns in the non-simulatable table (with total compiled size K). In practice all of these quantities are small for the boards the plugin targets, so all operations except the NGSpice call itself complete in well under a second.

Table 4.2: Worst-case complexity of the plugin's principal operations.

Operation	**Complexity**	**Remark**
Snapshot capture	$O(n)$	One pass over the board's footprints.
Snapshot write (JSON)	$O(n)$	Dominated by disk I/O at typical n.
Semantic diff	$O(n + m)$	Algorithm 4.1.
Schematic fingerprint	$O(\|\sigma\|)$	Linear pass for normalisation + SHA-256.
Stale check (one ckpt)	$O(1)$	After fingerprint is cached for the render.
Non-sim pre-flight	$O\big(K(n + \sum \|v_r\|)\big)$	One NFA pass per reference and value.
Timeline render	$O(T \cdot \bar{n})$	T rows, $\bar{n}$ average snapshot size.
Rollback	$O(n)$	Capture pre-rollback snapshot, then overwrite.
HTML export	$O(T \cdot \bar{n} + W)$	W is total waveform sample count.

The cost of an actual NGSpice simulation is outside the plugin's control and depends on the analysis type, the number of nodes, and the integration step size requested by the netlist. For the validation circuits in Chapter 7, all NGSpice runs completed in under five seconds on the development machine.

Chapter 5

Implementation: Core Plugin

This chapter describes the parts of the plugin that work regardless of whether the project can be simulated: snapshot capture, the semantic diff engine, the wxPython timeline, rollback, the per-component history view, and the HTML report. The mathematical foundations of these components have already been covered in Chapter 4. The simulation pipeline is treated separately in Chapter 6 because it has its own failure modes and depends on eSim and NGSpice being installed.

5.1 Architecture

The plugin is organised as a small Python package with one responsibility per module. Figure 5.1 shows the module dependency graph. The entry point (`__init__.py`) registers the plugin with KiCad. The hub module `plugin.py` owns the snapshot logic and is invoked by `board_listener.py` on the save event. The UI module `ui_panel.py` is the only consumer of `simulation_engine.py`, which is itself self-contained and imports no other plugin module. That isolation makes the simulation pipeline testable independently of the UI.

The plugin uses CPython 3.9 (the version bundled with KiCad 9.0) and imports only from the standard library and from wxPython [5], which KiCad already loads for its own UI. No `pip install` step is required. Development was carried out on Windows 11 for KiCad and on WSL2 Ubuntu for the supporting Python tooling. The plugin itself is platform-neutral, and all file operations go through `os.path.join` with UTF-8 encoding.

Table 5.1 summarises the role of each module in the plugin package.

5.2 Plugin Registration

The plugin registers with KiCad through the `ActionPlugin` mechanism. At startup, KiCad imports every Python module in its plugin directory and calls `.register()` on any class that inherits from `pcbnew.ActionPlugin` [2]. The registration code is small on purpose: it declares the plugin's metadata (name, category, description, toolbar visibility) and exposes a single `Run()` entry point that opens the timeline panel.

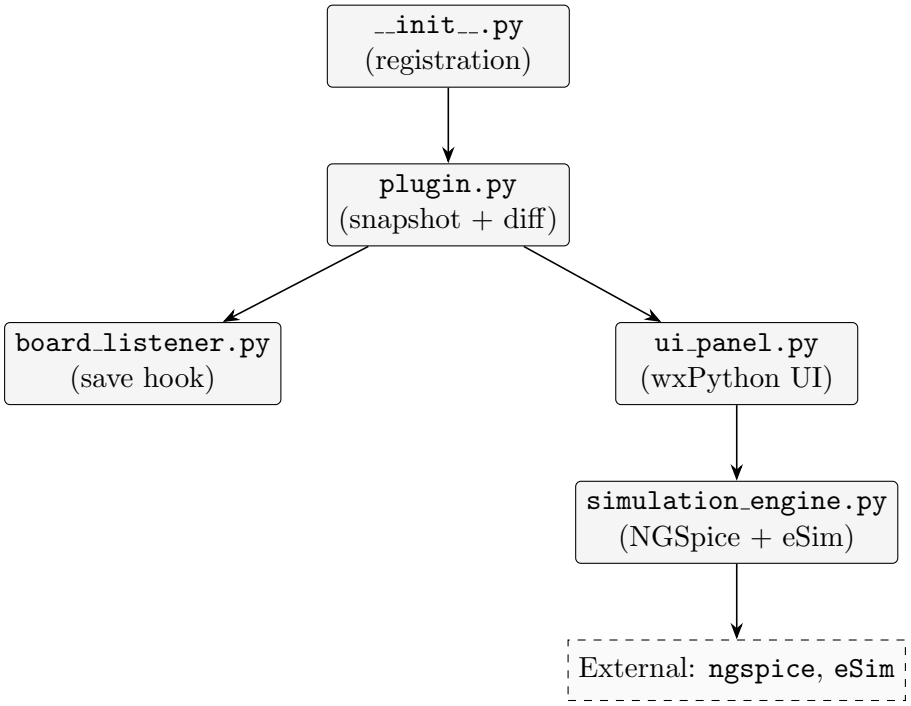

Figure 5.1: Module dependency graph. The dashed box represents external tools invoked through subprocess.

Table 5.1: Module responsibilities in the KiCad Design Diary plugin.

Module	**Responsibility**
`__init__.py`	Plugin registration with KiCad ActionPlugin mechanism.
`plugin.py`	Snapshot capture, semantic diff, history persistence.
`board_listener.py`	Save-event hook; invokes snapshot capture.
`ui_panel.py`	wxPython timeline panel, rollback, tag, export buttons.
`simulation_engine.py`	NGSpice invocation, raw-file parsing, non-simulatable detection.

The `Run()` method imports the UI module lazily rather than at the top of the file. This pattern is used so that any error in the UI module surfaces when the user clicks the toolbar button, rather than being swallowed during KiCad's startup scan of the plugin directory. Errors that surface at click time produce a normal Python traceback in KiCad's scripting console; errors that surface at startup time are hidden inside KiCad's startup log, which is harder to find.

After installation, the plugin appears under **Tools** → **External Plugins** as **KiCad Design Diary**, and clicking it opens the timeline panel described in Section 5.5.

5.3 Snapshot Capture

A snapshot is a JSON file holding the schematic and PCB state at a single moment in time. When the user saves the project, KiCad fires a board-save event, `board_listener.py` catches it, and `plugin.py` reads both the live PCB (through `pcbnew.GetBoard()`) and the `.kicad_sch` file (parsed directly as S-expressions, since KiCad does not expose a stable Python interface to the schematic at the symbol level).

The captured state is the mapping S_t defined in Section 4.1. For each footprint on the board, the plugin reads the reference designator, the symbolic value, the footprint library identifier, and the (x, y) position in millimetres, and writes them as a flat dictionary keyed by the reference. The full procedure is summarised in Algorithm 5.1.

Algorithm 5.1 Capture a snapshot of the live board state.

Require: Live `pcbnew.BOARD` object B, project path p, optional label ℓ.
Ensure: Snapshot file written to p/`.design_diary_<project>/`.

$S \leftarrow \{\,\}$ ▷ Empty mapping
for all footprint F in B.GetFootprints() **do**
  $r \leftarrow F$.GetReference()
  $v_r \leftarrow F$.GetValue()
  $f_r \leftarrow$ library item name of F
  $(x_r, y_r) \leftarrow$ position of F in mm
  $S[r] \leftarrow (v_r, f_r, x_r, y_r)$
end for
$t \leftarrow$ current UTC timestamp formatted as `YYYYMMDD_HHMMSS`
Write S together with ℓ as JSON to p/`.design_diary_<project>/`t`.json`

Snapshots live in a hidden `.design_diary_<project>/` folder at the project root. The folder also contains files with different prefixes, listed in Table 5.2. The UI uses these prefixes to filter snapshots by intent without having to open and parse each file.

Table 5.2: Files inside the `.design_diary_<project>/` folder.

Pattern	Meaning
`YYYYMMDD_HHMMSS.json`	Canonical project snapshot (board + schematic).
`SCH_YYYYMMDD_HHMMSS.json`	Schematic-only snapshot, parsed from `.kicad_sch`.
`RUN_YYYYMMDD_HHMMSS.json`	Simulation run record.
`sim_run_*.raw`	Raw NGSpice output.
`sim_plot_*.html`	Generated interactive waveform plot.
`design_diary_report.html`	Exported HTML report (single file, overwritten).
`_*.json`	Internal plugin state (warning markers, etc.).

5.4 The Semantic Diff Engine

After every snapshot, the plugin compares it to the previous snapshot using the procedure formalised in Section 4.2 (Algorithm 4.1). Three categories of change are detected: *added*, *removed*, and *modified*, with modified changes further decomposed into per-property change records.

The implementation is a pure function: it takes two dictionaries and returns a list of strings, with no I/O and no global state. This pattern makes the diff engine trivial to test in isolation, and trivial to reason about under the formal definitions of Chapter 4. There is no machine learning component; the templates that render each change record as English are fixed string templates, which keeps the engine fully deterministic and audit-able.

An alternative approach considered during development was to compare the raw text of the two snapshots and report line-level differences. This was easier to implement but harder to *read*: the user receives a hunk of S-expression syntax instead of a sentence. The semantic version is what makes the diary directly useful rather than merely present.

The timeline panel also exposes a *Compare Snapshots* button that lets the user pick any two snapshots from the list and view the full diff between them. This is useful for questions such as "what has changed since I tagged the board last Friday?" without scrolling through every intermediate diff.

5.5 Timeline Panel

The timeline panel is the user's entry point into the plugin. It is a single `wx.Frame` containing a `wx.ListCtrl` for the snapshot list, a `wx.TextCtrl` for the diff display, and a horizontal sizer for the action buttons. All UI logic lives in a single class, `DiaryPanel`, with one handler method per button. This flat structure was chosen deliberately: deeper UI hierarchies were tempting but would have been harder for a future intern to read.

wxPython was selected over PyQt or Tkinter for two practical reasons. First, wxPython is already loaded by KiCad's own UI, so the plugin adds no new de-

pendencies. Second, wxPython windows integrate cleanly with KiCad's window management and theming.

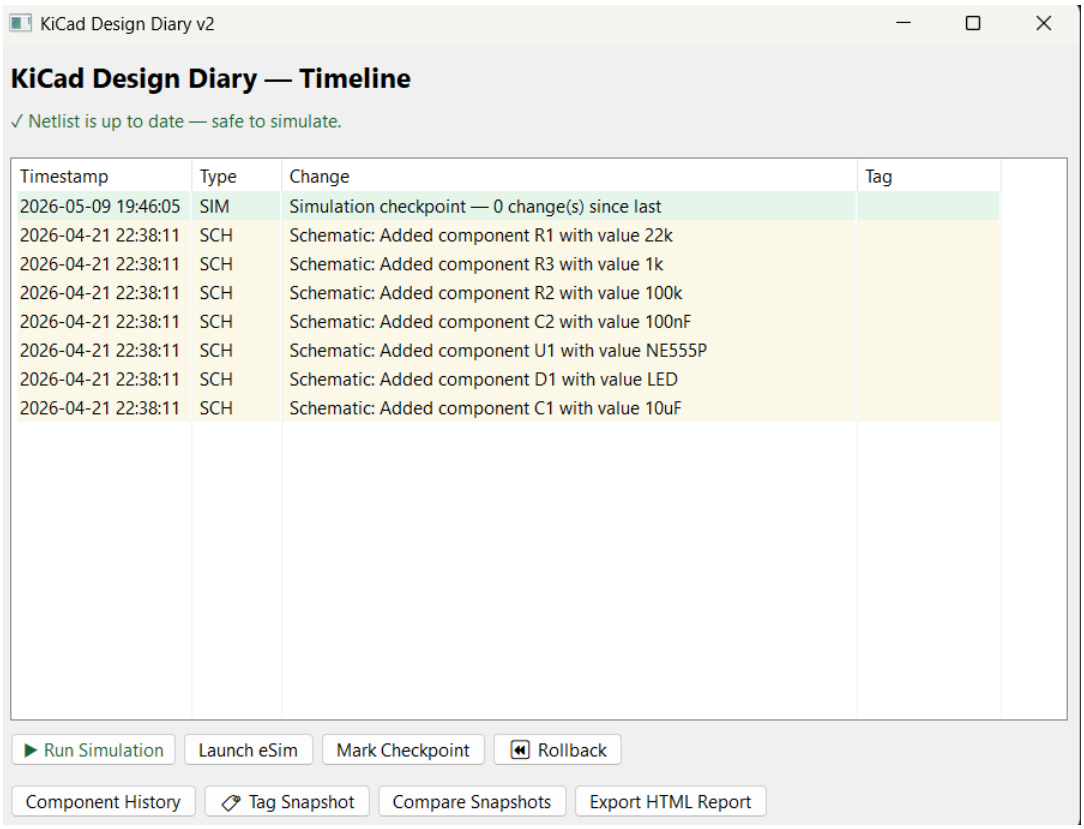

Figure 5.2: Timeline panel showing the schematic-build history of the NE555 validation circuit, with one simulation checkpoint at the top.

Each row in the panel shows a timestamp, a row type (`SCH`, `PCB`, `SIM`, or `ROLLB`), a change summary, and an optional user-supplied tag. Rows are colour-coded by type as described in Table 5.3. Selecting a row reveals the full diff in a lower pane. A status banner at the top reports whether the netlist on disk matches the current schematic.

Table 5.3: Row type colour coding in the timeline panel.

Type	**Colour**	**Meaning**
`SCH`	Cream	Schematic edit (added, removed, or modified component).
`PCB`	Light blue	PCB-level edit (footprint or position change).
`SIM`	Green	Simulation checkpoint with attached waveform.
`ROLLB`	Pink	Rollback event to an earlier snapshot.

Tag Snapshot. Any row can be tagged with a short label using the *Tag Snapshot* button. Tags appear in their own column and in the HTML report. They are

stored inside the snapshot JSON itself, so they survive across KiCad sessions and are included in the export.

5.6 Rollback

Rollback is exposed as a button at the bottom of the panel. When the user selects a snapshot and clicks *Rollback*, the plugin overwrites the live `.kicad_sch` (and `.kicad_pcb` where relevant) with the contents of the chosen snapshot. The procedure is summarised in Algorithm 5.2. Two design decisions are central.

Algorithm 5.2 Reversible rollback to an earlier snapshot.

Require: Target snapshot $S^\star$ chosen by the user; current board B; project path p.
Ensure: Live project files now match $S^\star$; a pre-rollback snapshot has been recorded in the diary folder.

1: Show confirmation dialog; abort if the user cancels.
2: $S_{\text{cur}} \leftarrow$ Algorithm 5.1 applied to B, with label *auto-saved before rollback*.
3: Write S_{cur} to the diary folder.
4: Overwrite `.kicad_sch` and `.kicad_pcb` under p using the contents of $S^\star$.
5: Call `pcbnew.Refresh()` so KiCad reloads the board.
6: Refresh the timeline panel.

First, the plugin captures a fresh snapshot of the current state *before* the rollback overwrites it, labelled *auto-saved before rollback*. This makes rollback itself reversible. This safeguard was added after testing the first version, where one misclicked rollback destroyed work. A version-control tool that loses information is worse than no tool.

Second, rollback does not compute a reverse patch from the diff engine. An early prototype attempted this, inverting the diff and applying it to the live file. This approach was rejected for two reasons. Any property the diff engine misses becomes a silent corruption on rollback. And when a rollback gives the wrong result, it is not clear whether the bug is in the diff engine, the patch inverter, or the patch applier. Overwriting the file with the historical snapshot is deterministic and obviously correct. The cost is some redundant disk traffic, which is negligible at the file sizes KiCad produces.

5.7 Component History

Component History is a per-component view of the timeline. The user selects a reference designator such as `R1` or `U1`, and the plugin walks the full snapshot history and produces a chronological list of every value, footprint, or position change for just that component. This is much faster to scan than the full timeline when the question is "what has happened to capacitor C2 over the life of this project?"

Internally, the view replays the snapshot list and filters each diff record for entries whose reference matches the chosen designator. It also feeds into the *Modification Frequency* chart in the HTML report (Section 5.8), where each component's total

change count is rendered as a horizontal bar. Components that have been edited many times are visible at a glance, which is useful for identifying parts of a design that are proving troublesome.

5.8 HTML Report

The export format is a single self-contained HTML file with all CSS, JavaScript, and embedded waveform data inlined. The report can be emailed, uploaded, or attached to a project submission without the risk of broken links or missing assets. HTML was chosen over PDF because PDF generation from inside a plugin process introduces extra dependencies (typically a headless browser or a LaTeX engine), and any user with a web browser can open HTML.

Table 5.4 lists the sections included in the report.

Table 5.4: HTML report sections.

Section	Description
Header and statistics	Board name, generation timestamp, netlist status, four summary cards.
Modification frequency	Horizontal bar chart of edit counts per component reference.
Simulation history	One card per simulation checkpoint, with stale/current indication.
Change timeline	Chronological event list with type-specific badges.
Embedded waveform plots	Interactive plot per simulation checkpoint with toggleable signals.

Header and statistics (Figure 5.3). A banner identifies the board, the generation timestamp, and the netlist-up-to-date status. Below it, four summary cards give the project's total sessions, total change events, distinct component references tracked, and simulation run count.

Modification frequency (Figure 5.4). Each component reference is plotted as a horizontal bar whose length is the number of times that component has been modified across the project's history.

Simulation and checkpoint history (Figure 5.5). Every simulation run appears as a coloured card with the timestamp on the left and the outcome on the right. Stale checkpoints are distinguished from current ones by background colour.

Change timeline (Figure 5.6). The full chronological event list, with type-specific pill badges (`SCHEMATIC`, `SIMULATION`, `ROLLBACK`), in the same colour scheme as the in-KiCad timeline panel.

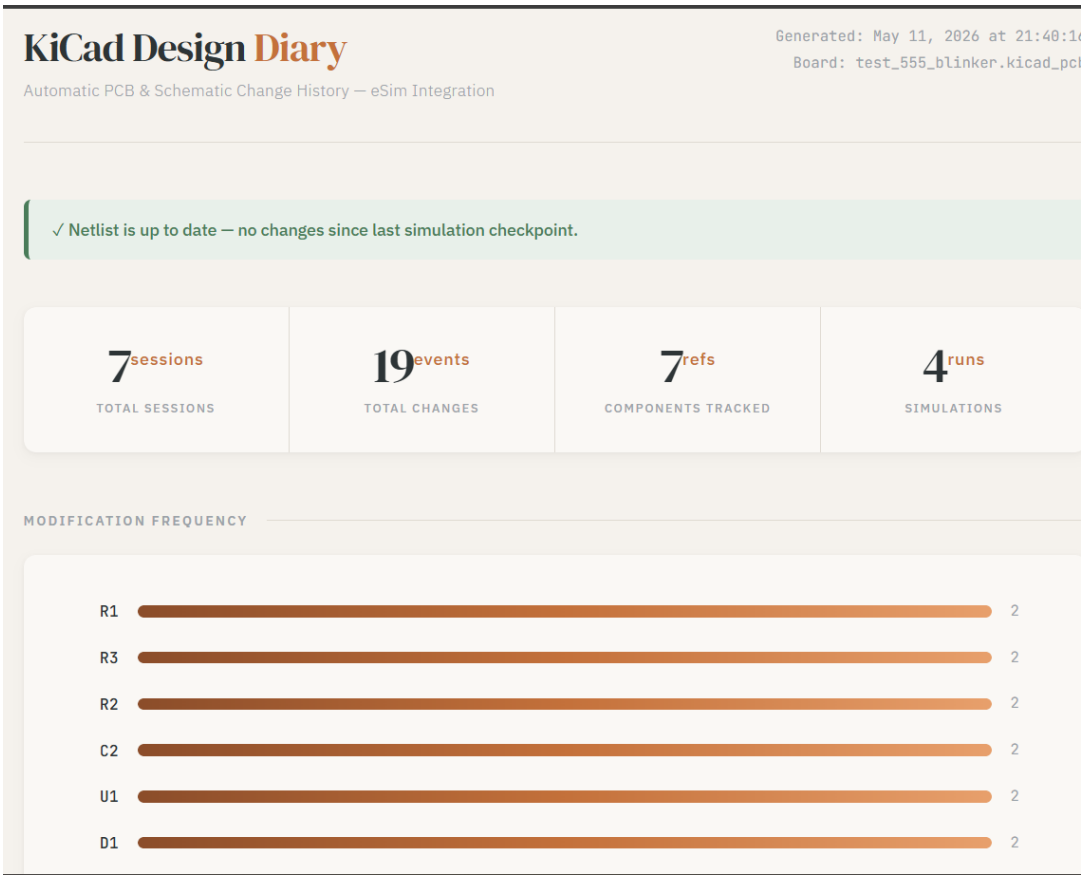

Figure 5.3: HTML report header for the NE555 board: green status banner and four summary cards.

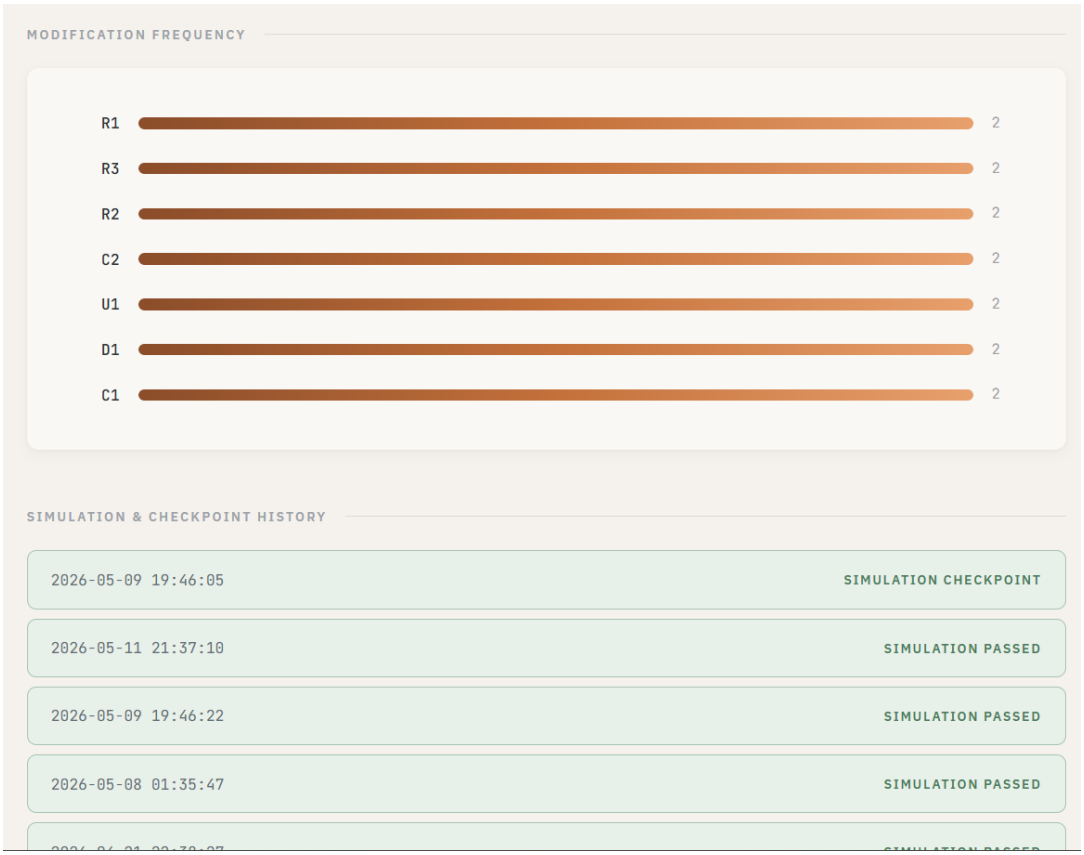

Figure 5.4: Modification frequency chart: one bar per component reference, with bar length proportional to edit count.

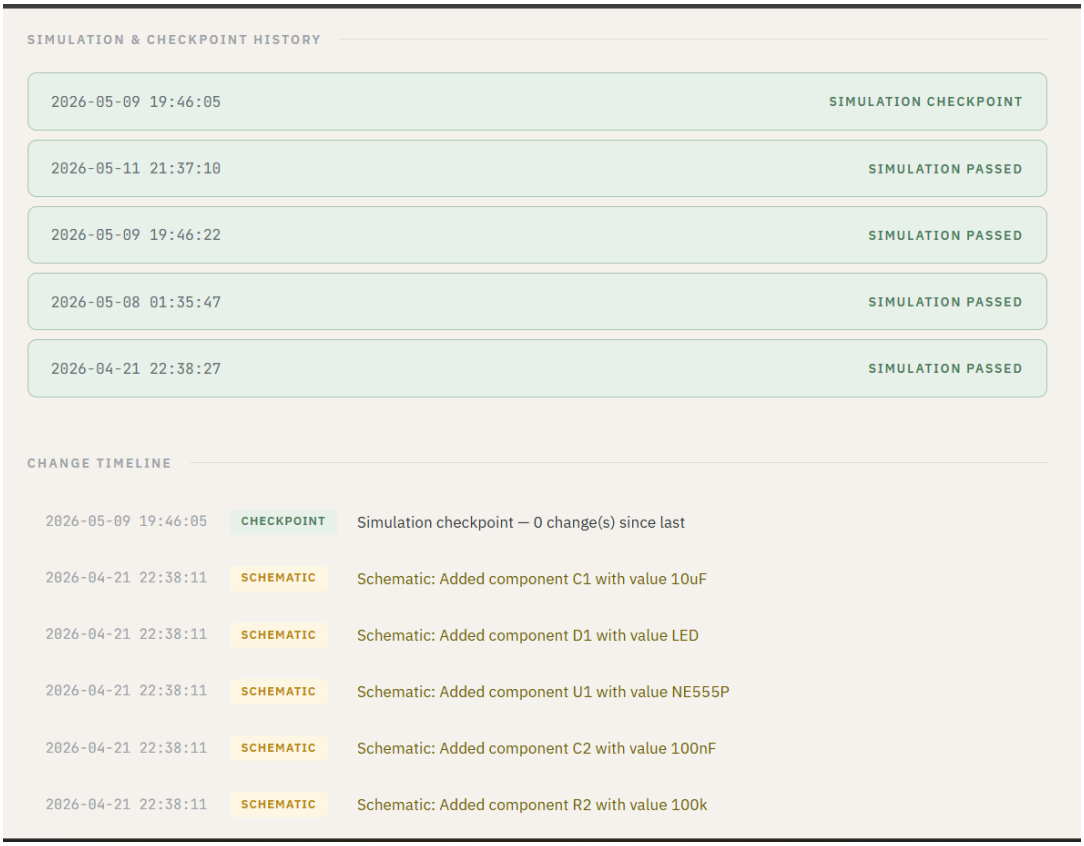

Figure 5.5: Simulation history section. Each card shows a timestamp and an outcome badge.

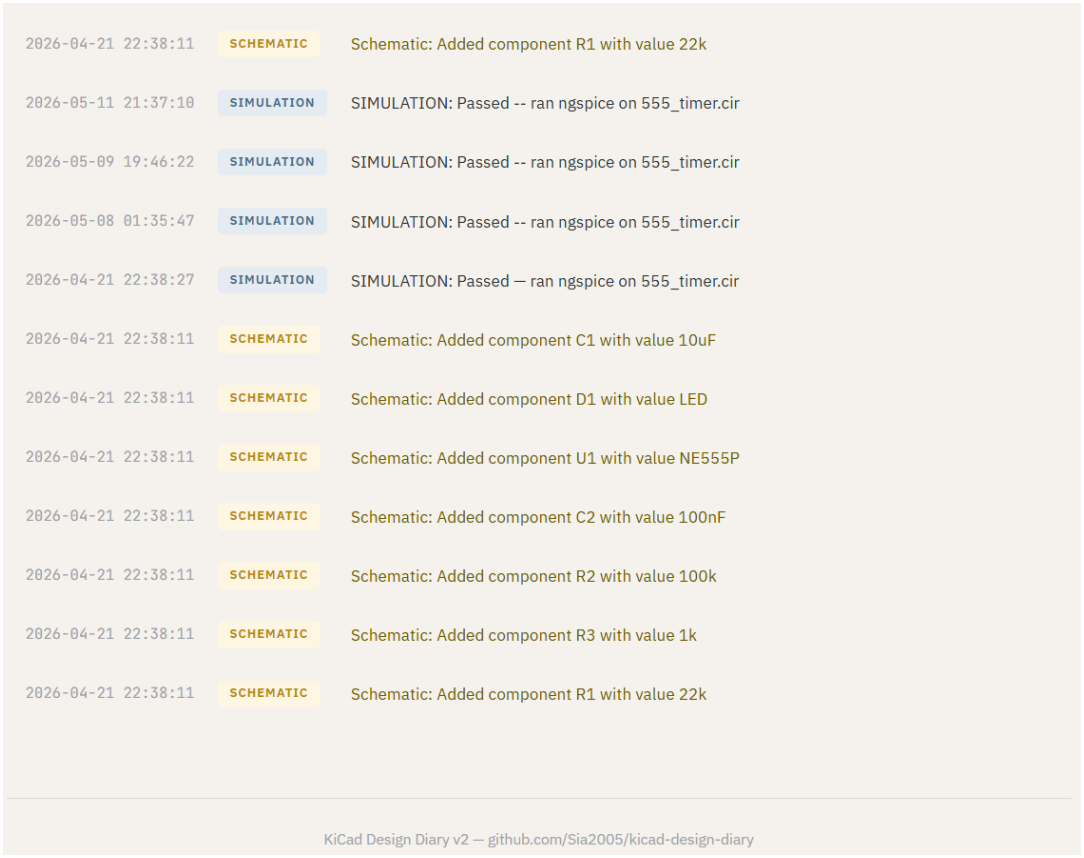

Figure 5.6: Change timeline in the HTML report: timestamp, type badge, one-line description per event.

Chapter 6

Implementation: Simulation Checkpointing with eSim and NGSpice

A simulation checkpoint is a snapshot with three additional artefacts attached: the SPICE netlist used for the simulation, the NGSpice raw output file, and a SHA-256 hash of the schematic at the time of simulation. This chapter covers how the checkpoint is constructed, how the plugin detects when a checkpoint has gone stale (using the fingerprint mechanism formalised in Section 4.3), how non-simulatable circuits are caught before NGSpice runs (using the regular-language formulation of Section 4.4), and how the remaining NGSpice errors are translated into actionable messages.

6.1 The Run-Simulation Decision

When the user clicks *Run Simulation*, the plugin first determines whether the board is simulatable. The pattern check is fast (a small number of regular expressions are applied to the component dictionary), and it saves the user from a wall of cryptic SPICE errors on boards that NGSpice was never going to handle. The decision tree is shown in Figure 6.1.

6.2 Invoking eSim's Converter and NGSpice

For a simulatable board, the plugin invokes eSim's KiCad-to-NGSpice converter [3] to produce a netlist, then runs NGSpice against it in batch mode [4]. The converter is not re-implemented inside the plugin. eSim already performs this work, and a parallel implementation would slowly drift from eSim's conventions as eSim itself evolves. Anyone using this plugin already has eSim installed, since they need it for the rest of the FOSSEE simulation flow, so calling out to eSim's converter does not introduce a new dependency in practice.

NGSpice is invoked in batch mode (the `-b` command-line flag), the same way eSim itself uses it. Batch mode causes NGSpice to read the netlist, run the analysis

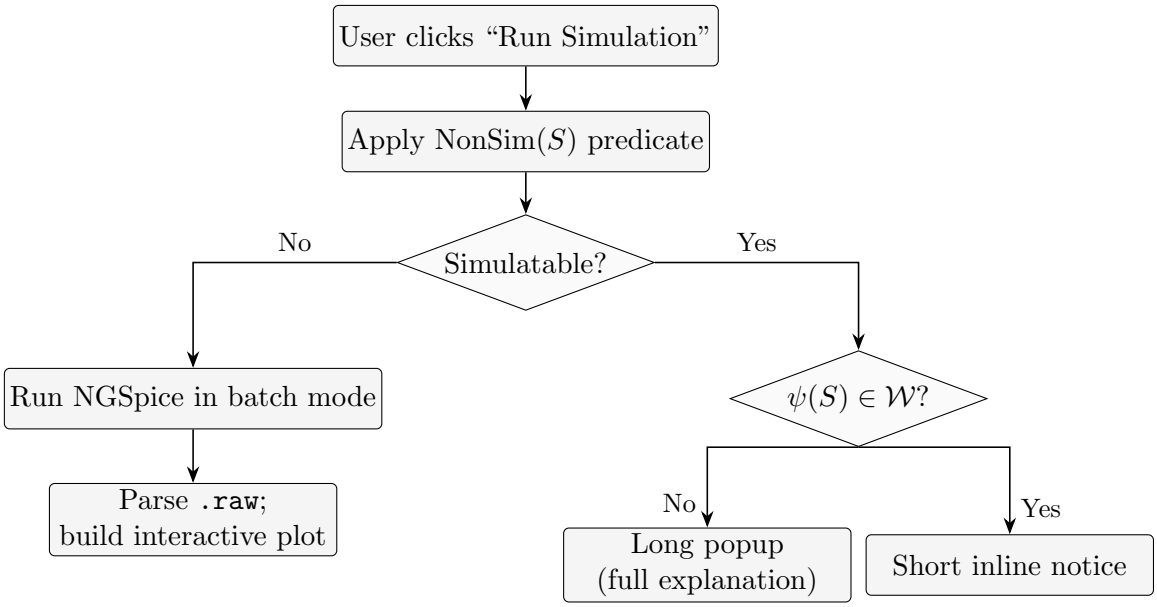

Figure 6.1: Decision tree for the “Run Simulation” action. The predicate NonSim is defined in Section 4.4; the fingerprint ψ and the warned-fingerprint set $\mathcal{W}$ are defined in Section 4.5.

directives it contains, write results to a raw file at the path supplied via the `-r` flag, and exit. There is no interactive prompt, no plot windows, and no terminal emulator involved. The plugin captures the process's standard output and standard error, persists them to a log file, and inspects the return code to decide whether the simulation succeeded. An earlier FOSSEE intern arrived at the same choice for the same reason [9], and reading that report helped confirm the approach.

The end-to-end simulation procedure is summarised in Algorithm 6.1.

Algorithm 6.1 Simulation pipeline triggered by "Run Simulation".

Require: Current snapshot S, schematic bytes σ, project path p.
Ensure: Either (a) checkpoint written and waveform rendered, or (b) user shown the appropriate non-sim notice, or (c) error translated and surfaced.

if NonSim(S) **then**
    Compute warning fingerprint $\psi(S)$.
    if $\psi(S) \notin \mathcal{W}$ **then**
        Show the long popup; add $\psi(S)$ to $\mathcal{W}$.
    else
        Show the short inline notice.
    end if
    return
end if
Invoke the eSim converter on p to produce a `.cir` netlist.
Run NGSpice in batch mode with the `-b -r` flags; capture stdout, stderr, return code.
if NGSpice returned non-zero **then**
    Pass stdout and stderr to the error translator (Section 6.5); surface the resulting message.
    return
end if
Parse the `.raw` file; build the interactive plot.
Compute the schematic fingerprint $\phi(\sigma)$ (Section 4.3).
Record a new snapshot with the simulation artefacts and ϕ attached.
Refresh the timeline panel.

Once NGSpice exits, the plugin parses the raw output into an interactive waveform plot. Each signal in the raw file becomes a toggleable checkbox at the top of the page; selecting or deselecting a checkbox adds or removes that signal from the chart in real time. Figure 6.2 shows the plot view from the Sallen–Key validation in Chapter 7.

The first version of this pipeline used a `wx.Timer` polling loop to detect when NGSpice had finished writing the raw file. This was inefficient and produced visible UI sluggishness on slower machines. It was replaced with a synchronous `subprocess` call inside a worker thread, which simply waits for NGSpice to exit and signals the UI thread to refresh. The revised version is both faster and easier to reason about.

6.3 Stale Checkpoint Detection

The schematic hash recorded at simulation time is the value $\phi(\sigma)$ from Section 4.3. Each time the timeline is rendered, every checkpoint's stored hash is compared against the current schematic hash ϕ_{now}. A mismatch (i.e. Stale($\phi_{\text{sim}}, \phi_{\text{now}}$) holds) flags the checkpoint as stale in both the timeline panel and the HTML report, with a short note about which schematic elements have diverged. The user can re-run the simulation against the current state, which creates a new checkpoint that supersedes

Figure 6.2: Interactive waveform view after a successful NGSpice run (Sallen–Key validation).

the stale one without removing it from the history. Stale checkpoints remain visible in the record but are clearly flagged.

The whitespace normalisation in the hash function is what makes the detector robust across operating systems. Without normalisation, line-ending differences between Windows (CRLF) and Linux (LF) would produce false-positive stale flags on projects opened on both systems. This was an actual problem during development on the WSL/Windows split, and the normalisation step resolved it.

SHA-256 is more than is strictly required for a non-cryptographic use case, but it is already available in the Python standard library and the cost is negligible at the file sizes KiCad produces. A faster hash such as xxHash would have been worthwhile only if the schematic files were substantially larger, which is not the case in practice.

6.4 Non-Simulatable Circuit Detection

Not every KiCad schematic can be simulated by NGSpice. Boards built around microcontrollers, programmers, EEPROMs, digital logic chips, or boards that consist mostly of connectors and headers fall outside what SPICE can model without custom subcircuit libraries. Invoking NGSpice on such a board produces a wall of cryptic errors that are difficult to act on.

The plugin addresses this by applying the predicate $\mathrm{NonSim}(S)$ defined in Section 4.4, with the regular-expression table $\mathcal{T}$ listed in Table 4.1. Before invoking NGSpice, the captured component dictionary is run through the patterns. If the

predicate fires, simulation is not started, and the user is shown a plain-language explanation that lists the specific components that triggered the warning.

To avoid showing the same long popup every time the user clicks *Run Simulation* on a board the plugin has already flagged, the plugin maintains the warned-fingerprint set $\mathcal{W}$ from Section 4.5, persisted as `_sim_warning_shown.json` in the diary folder. Repeated clicks on the same circuit show only a short inline notice instead of the long explanatory popup, because the fingerprint of the offending references is already in $\mathcal{W}$.

Adding a new pattern to $\mathcal{T}$ requires no other code changes; the union $\mathcal{L}_{\text{NS}}$ is automatically extended and remains regular, as discussed in Section 4.4. This makes the plugin easy to maintain in production: when a user reports that a new chip family is being misidentified, a single row is added to the table and the fix ships in the next release.

6.5 NGSpice Error Translation

When NGSpice fails on a board that the pattern table considered simulatable, the plugin parses the error output and translates the most common failure modes into actionable messages. The translator is a small lookup procedure: the combined stdout and stderr are searched for known error fragments, and the first matching fragment is mapped to a plain-language explanation. If no fragment matches, the translator returns nothing and the caller falls through to a generic failure message that includes the raw stderr.

Table 6.1 lists the failure modes currently recognised. The same convention as the pattern table applies: extending the translator means adding a row.

Table 6.1: NGSpice failure modes recognised by the error translator.

NGSpice fragment	**Plain-language explanation surfaced**
`unknown subcircuit`	A subcircuit definition is missing. Check that all `.lib` and `.include` directives in the `.cir` file resolve to files that exist on disk.
`singular matrix`	NGSpice reported a singular matrix during DC analysis. This usually means a floating node, a missing ground, or a zero-resistance loop.
`no convergence` or `iteration limit`	NGSpice could not converge. Try increasing the iteration limit, simplifying the circuit, or checking for very large or very small values.

6.6 Engineering Constraints Encountered

Two practical issues shaped the simulation pipeline during development.

The first was **path handling**. The development project was stored inside a OneDrive-synchronised directory whose path contained spaces. NGSpice on Windows does not handle spaces in file paths reliably. The resolution was to quote all paths in the NGSpice invocation and, where that proved insufficient, copy the netlist to a path-safe staging area under `%TEMP%` before running.

The second was **KiCad's scripting console**. On Windows, the console does not accept multi-line paste input, so every exploratory debugging session had to be entered one line at a time. This influenced how the plugin's debugging code is structured: as small individually-runnable functions, not multi-line scripts.

Chapter 7

Test Circuits and Results

The plugin was validated end-to-end on three circuits, chosen to exercise three different code paths in the simulation pipeline and to cover the full feature set of the plugin.

- An **NE555 astable multivibrator**, built from scratch, to test transient simulation.
- The bundled **Sallen–Key low-pass filter** from KiCad's simulation demos, to test AC analysis on a circuit with external SPICE library dependencies.
- A **PIC microcontroller programmer** board, to test the non-simulatable detection path.

Together, these three exercise transient simulation, AC analysis, external library handling, the rollback mechanism, the stale-checkpoint detector, the non-simulatable detector, and the HTML export. Table 7.1 summarises the outcome of each test.

Table 7.1: Summary of results across all test circuits.

Circuit	**Analysis**	**Outcome**	**Features Exercised**
NE555 astable	Transient	Simulated	Snapshot, diff, sim, stale flag, rollback, HTML
Sallen–Key LPF	AC sweep	Simulated	External SPICE library, AC analysis, rollback
PIC programmer	N/A	Non-sim	Non-simulatable detection, fallback to history-only

7.1 Test Circuit 1: NE555 Astable Multivibrator

7.1.1 Circuit Description

The first circuit is an NE555 astable multivibrator built in KiCad. The schematic (Figure 7.1) is a standard arrangement: NE555P (U1), timing resistors R3 and R4, timing capacitor C1, bias divider R1/R2, decoupling capacitor C2, and an LED indicator (D1) through current-limiting resistor R5. This circuit was chosen as the primary validation case because it is small enough to inspect at a glance, produces a recognisable output that is easy to verify visually, and has component values that change the output in obvious ways (changing a timing resistor changes the duty cycle), which makes it a clear test of stale-checkpoint detection.

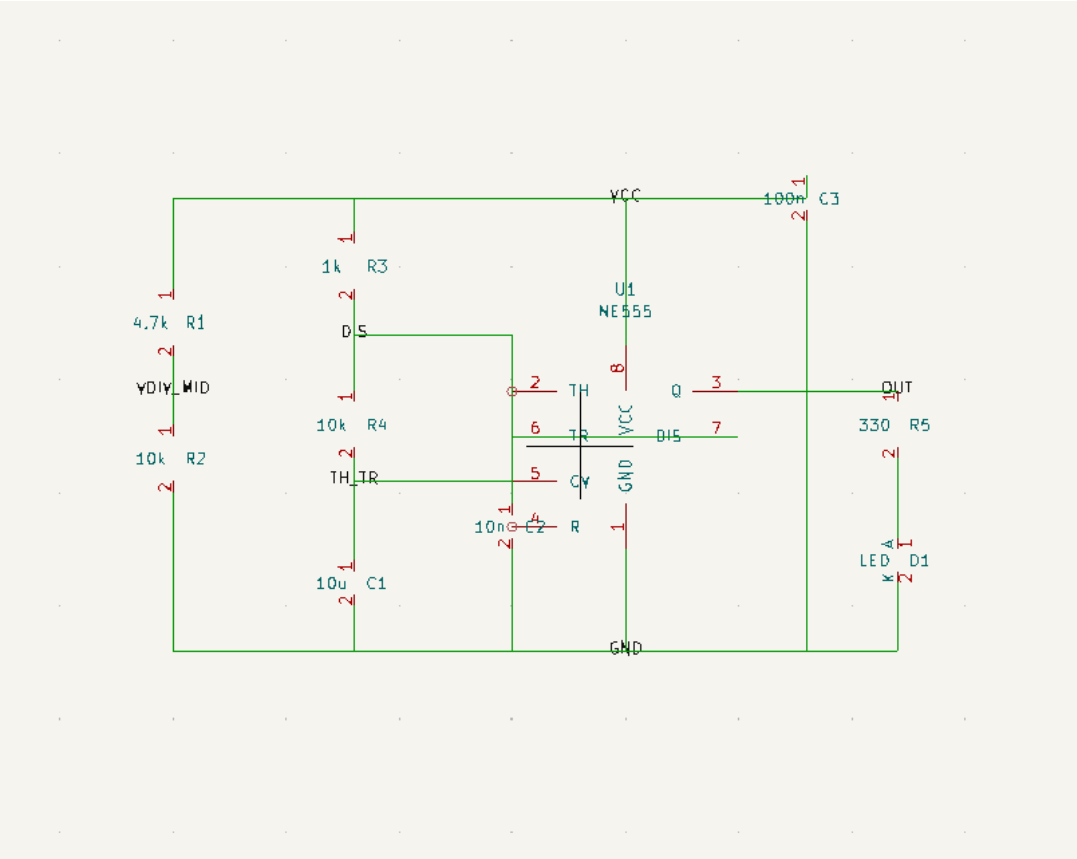

Figure 7.1: NE555 astable multivibrator schematic, used as the primary validation circuit.

7.1.2 Validation Procedure and Results

The validation procedure was as follows.

1. The schematic was built up component by component, with the plugin installed. Each save produced a separate `SCH` entry in the timeline (Figure 5.2), one per added component.

2. Clicking *Run Simulation* on the completed schematic invoked eSim's converter, produced `555_timer.cir`, and ran NGSpice in batch mode. The resulting transient waveform is shown in Figure 7.2: `v(out)` sits near the supply rail, `v(chg)` traces the RC charging curve through the timing capacitor, and `v(led_a)` sits at the forward drop of the LED. The simulation completed with no warnings and was recorded as a `SIM` entry. The status banner reported *Netlist is up to date — safe to simulate.*

3. A timing resistor's value was modified in the schematic editor and saved. The diff engine reported a single change, *"Changed R1 value: . . . "*, with no spurious modifications elsewhere on the board, and the previously recorded simulation checkpoint was marked as stale (the predicate $\text{Stale}(\phi_{\text{sim}}, \phi_{\text{now}})$ from Section 4.3 fired). The netlist-up-to-date banner cleared.

4. A rollback to an earlier snapshot was triggered through the timeline panel. The resistor value reverted. An *auto-saved before rollback* snapshot appeared in the timeline immediately above the rollback target, and the status banner returned to its up-to-date state.

5. The HTML report was exported. All snapshots rendered chronologically, all diffs displayed correctly, and the embedded waveform plot opened with full interactivity in the browser. The header section (Figure 5.3) recorded 7 sessions, 19 change events, 7 component references, and 4 simulation runs.

All five steps completed correctly. The end-to-end behaviour of every core plugin feature (automatic capture, semantic diff, simulation, stale detection, rollback, and HTML export) was verified on a single real circuit before moving to the remaining validation cases.

7.2 Test Circuit 2: Sallen–Key Low-Pass Filter

7.2.1 Circuit Description

The second circuit is the `sallen_key` project from KiCad's bundled simulation demos [10], located at `share/kicad/demos/simulation/sallen_key/` in a default installation. The schematic (Figure 7.3) is a single-stage Sallen–Key low-pass filter built around an AD8051 op-amp model, with $R_1 = R_2 = 1\,\text{k}\Omega$, $C_1 = 110\,\text{nF}$, $C_2 = 100\,\text{nF}$, an AC source V1 at the input, and a `.ac dec 10 1 1Meg` directive specifying a frequency sweep from 1 Hz to 1 MHz with 10 points per decade.

This circuit exercises a different class of NGSpice analysis from the NE555 (AC sweep rather than transient), and depends on an external SPICE library (`ad8051.lib`) included with the demo, which makes it a more realistic test of eSim's converter handling of external models. As a KiCad bundled demo, it is also reproducible: any future intern can install KiCad, open the same project, and verify the plugin's behaviour against the same reference circuit.

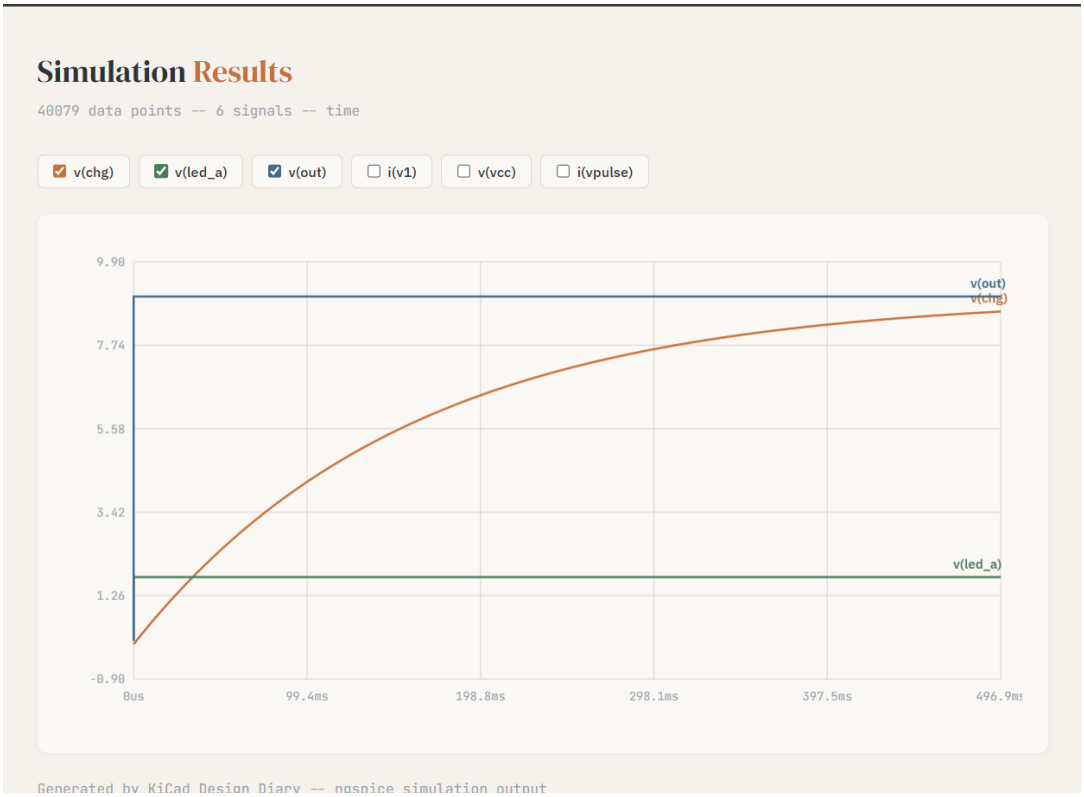

Figure 7.2: NE555 simulation result. Orange `v(chg)`: timing capacitor charging. Blue `v(out)`: timer output. Green `v(led_a)`: LED forward drop.

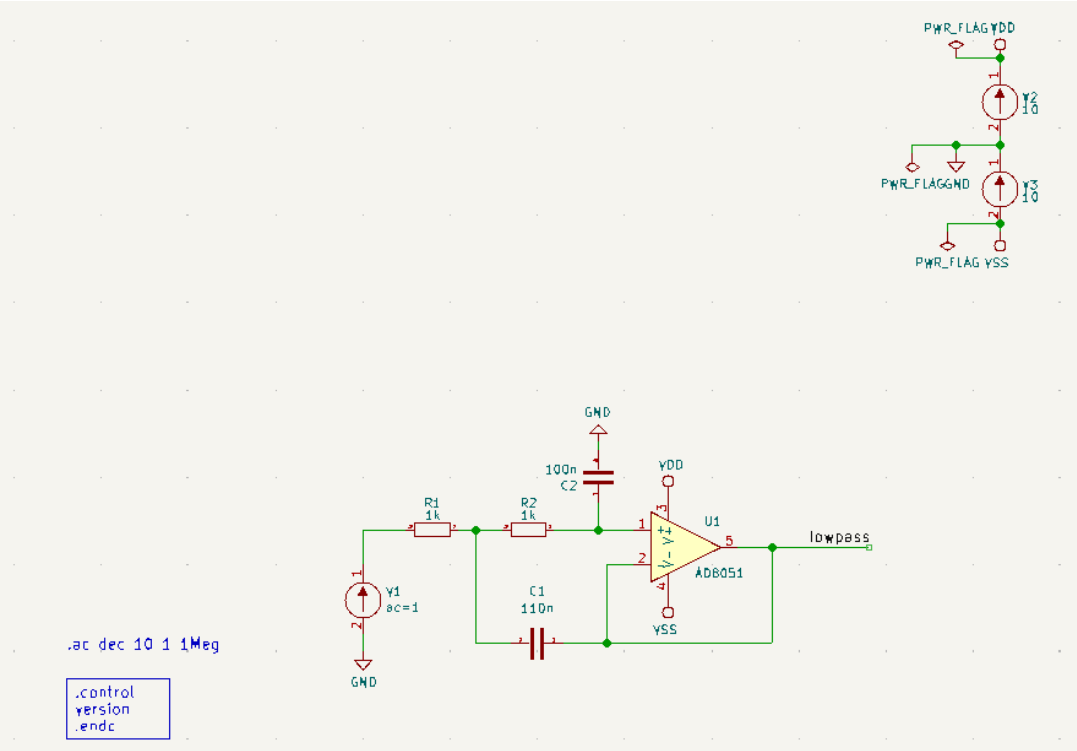

Figure 7.3: Sallen–Key low-pass filter from the bundled KiCad demos, used to test AC analysis.

7.2.2 Validation Procedure and Results

The procedure was as follows.

1. The `sallen_key` demo was opened and saved once to register a baseline snapshot.
2. Clicking *Run Simulation* produced a netlist, ran the AC sweep, and rendered the result in the interactive viewer (Figure 6.2, in Chapter 6). Six signals were captured: `i(e1)`, `v(in)`, `v(n1)`, `v(n2)`, `v(out)`, and `i(v1)`. Any subset of these can be toggled for display.
3. One of the filter resistors was modified, the schematic was saved, and the simulation was re-run. The previous checkpoint became stale, a fresh AC sweep was recorded against the new schematic, and both entries appeared in the timeline.
4. A rollback to the original Sallen–Key configuration was performed. The timeline records this as a pink `ROLLB...` row, with an auto-saved snapshot of the pre-rollback state immediately above it. The full sequence of three SIM checkpoints and two ROLLBACK events is visible in Figure 7.4.

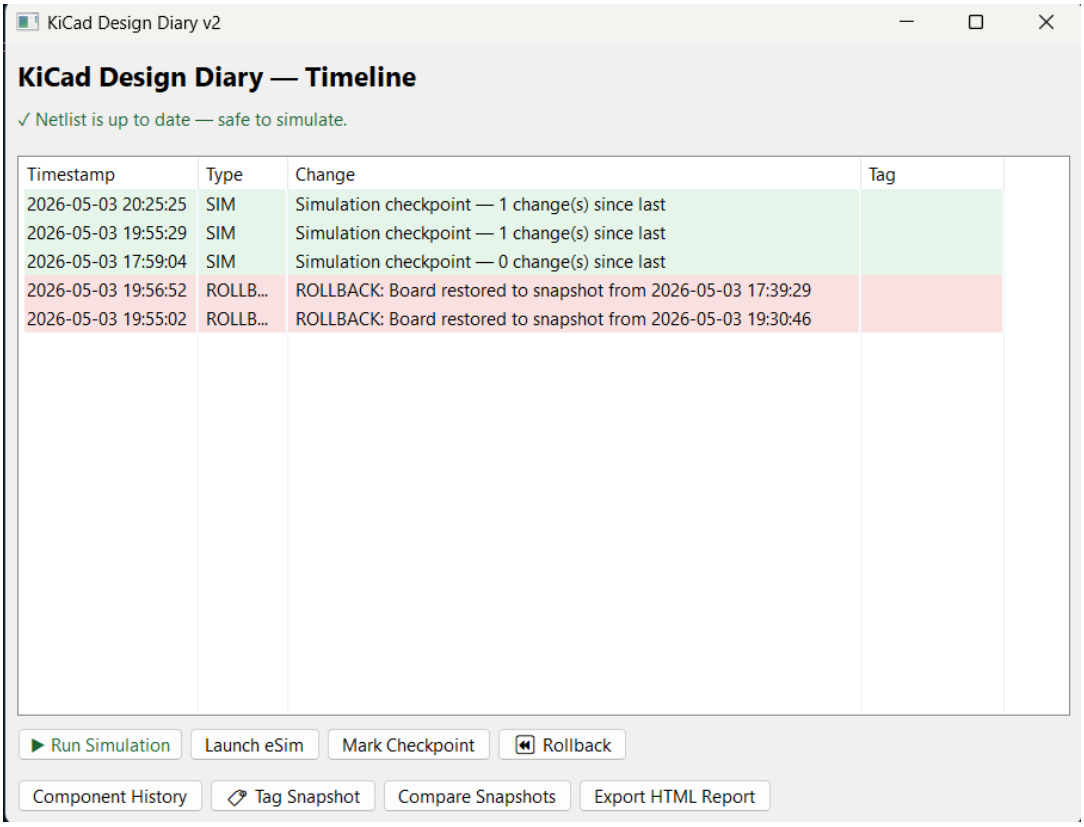

Figure 7.4: Sallen–Key timeline: three SIM checkpoints (green) interleaved with two ROLLBACK events (pink).

All four steps completed correctly. The Sallen–Key case confirms that the plugin handles an NGSpice analysis type different from the NE555 and that rollback behaves correctly on a circuit with external SPICE library dependencies.

7.3 Test Circuit 3: PIC Microcontroller Programmer

7.3.1 Circuit Description

The third circuit is a PIC microcontroller programmer (a JDM-COMB4 programmer with a 13 V DC/DC converter section). The schematic (Figure 7.5) is a mixed-signal board with a PIC socket, 74HC125 logic, BC237/BC307 transistors, an LT1373 boost regulator for the 13 V programming voltage, a DB9 serial connector, and a large number of pin-header connectors (P101–P106 and others).

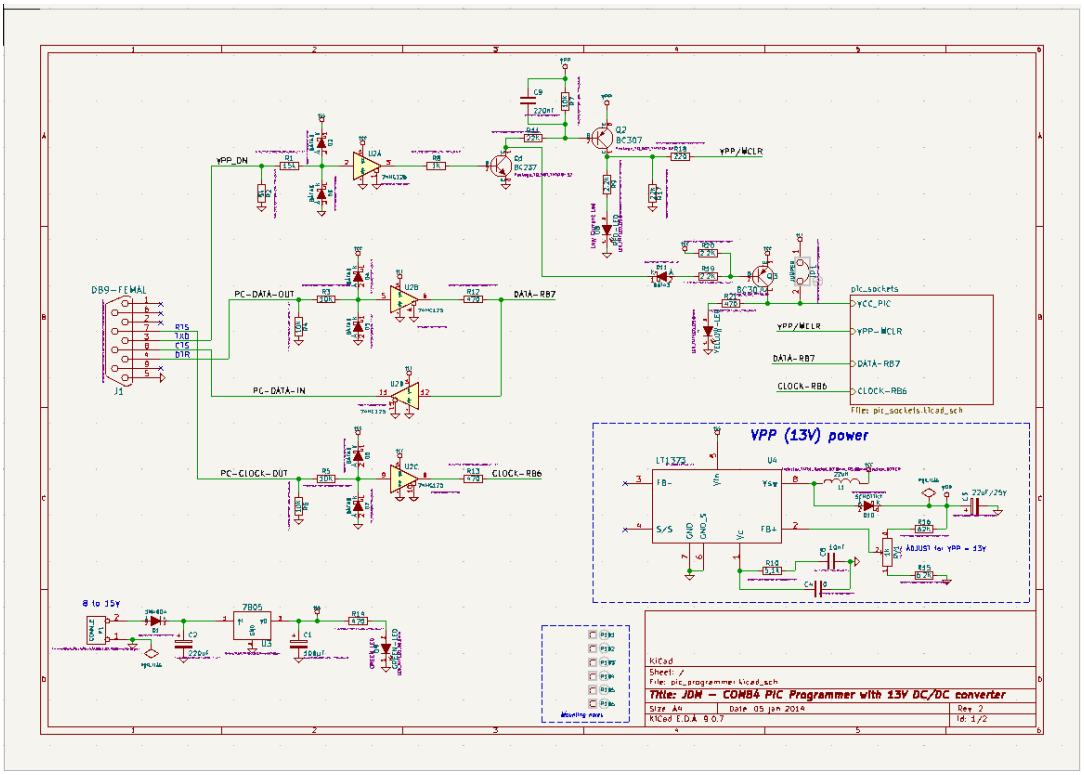

Figure 7.5: PIC programmer schematic, used to test the non-simulatable detection path. None of the highlighted parts can be modelled in NGSpice without custom subcircuit libraries.

The expected behaviour on this board is *not* to produce a simulation. It is to detect that the circuit is non-simulatable and explain to the user why no simulation can be run. Formally, the predicate NonSim(S) from Section 4.4 should fire on several references at once: the PIC socket, the 74HC logic, and the P-prefixed connectors.

7.3.2 Validation Procedure and Results

The procedure was as follows.

1. The PIC programmer schematic was opened in KiCad and saved to register a baseline snapshot. Subsequent edits, including a value change to capacitor C2

(220 μF to 210 μF) and two successive rollback events, were recorded automatically in the timeline (Figure 7.6).

2. *Run Simulation* was clicked. The non-simulatable detection routine ran the component dictionary through the pattern table $\mathcal{T}$, identified the connectors `P101`, `P102`, `P103` and 13 further non-simulatable references, and presented the popup in Figure 7.7: *"Not simulatable in NGSpice — contains: P101, P102, P103 +13 more. All other Design Diary features still work."* The wording emphasises that snapshot, diff, rollback, tag, and HTML-export features remain fully functional even though simulation is disabled.

3. *Run Simulation* was clicked a second time. The fingerprint $\psi(S)$ from Section 4.5 was already in $\mathcal{W}$ (persisted as `_sim_warning_shown.json`), so the plugin presented a short inline notice instead of the long popup.

4. No netlist generation occurred, no NGSpice invocation happened, and no `.raw` file was produced. The timeline correctly reflected this: no `SIM` entries were created for this circuit, while `PCB` and `ROLLBACK` rows continued to be recorded normally.

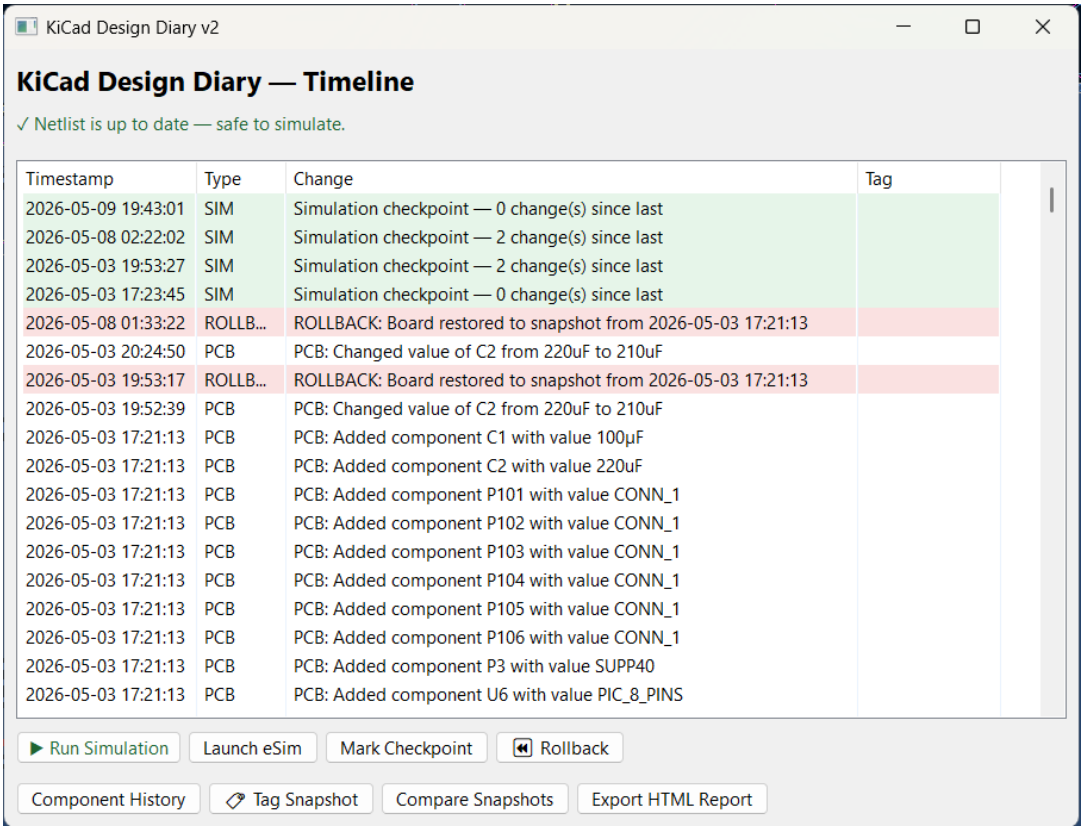

Figure 7.6: PIC programmer timeline: two pink `ROLLB...` rows and `PCB` edit rows including the C2 change. No new SIM entries created on this non-simulatable board.

The PIC case confirms that the plugin handles non-simulatable circuits gracefully. The user receives a clear explanation instead of a wall of SPICE errors, and the

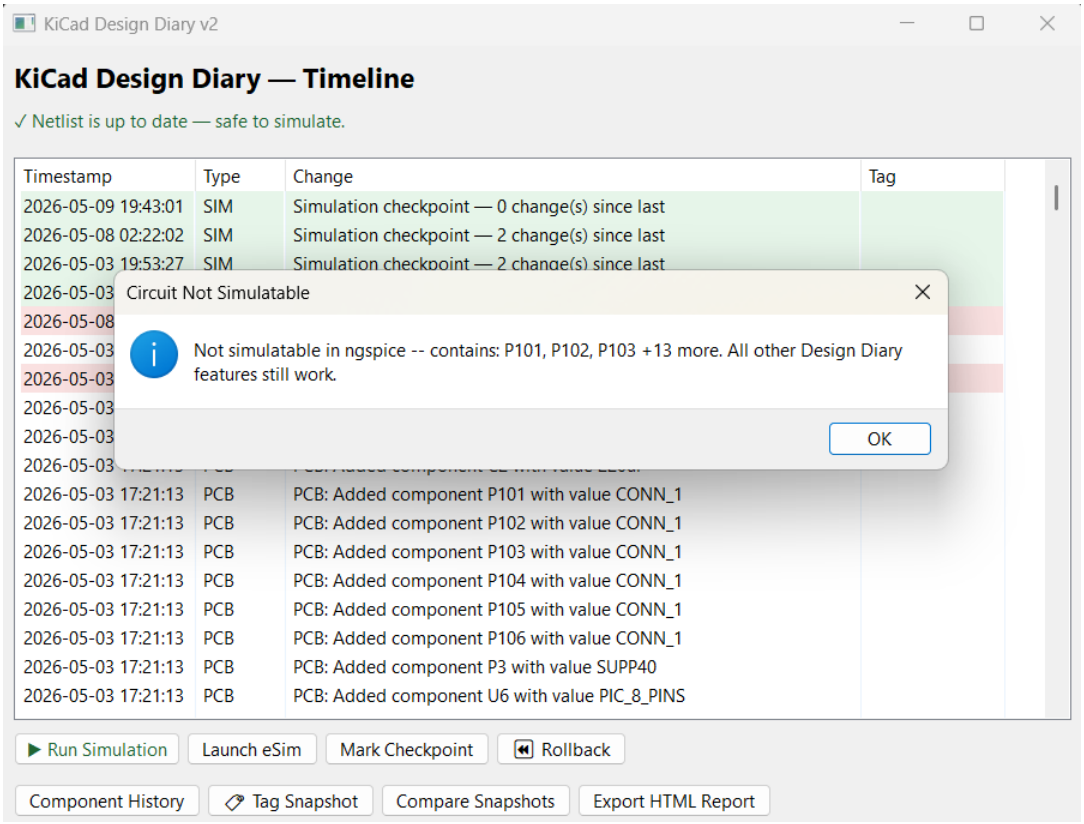

Figure 7.7: The non-simulatable popup: lists offending components and confirms that the rest of the plugin still works.

rest of the plugin (snapshots, rollbacks, HTML export) continues to work normally on the same project.

Chapter 8

Conclusion and Future Scope

8.1 Conclusion

This project developed a KiCad ActionPlugin (KiCad Design Diary) that fills a real gap in the open-source EDA workflow: there has not been an integrated, semantic, configuration-free design-history tool for KiCad and eSim users until now. The plugin captures snapshots automatically on save, computes a semantic diff between them, displays a native wxPython timeline inside KiCad, supports rollback to any earlier state, runs NGSpice simulations through eSim and binds each waveform to the schematic that produced it, marks waveforms as stale when the schematic changes underneath them, detects non-simulatable circuits before invoking NGSpice, exposes a per-component history view, and exports the whole record as a single self-contained HTML file.

The plugin is built on small, plain mathematical objects: snapshots as finite mappings, diffs as set-theoretic differences, fingerprints as SHA-256 hashes over a normalised schematic, and non-simulatable detection as a finite union of regular languages. Writing these structures down explicitly in Chapter 4 was useful in its own right: it forced the implementation to be checked against a specification, and it gives the next intern a precise statement of what each component is supposed to do.

End-to-end validation on three circuits (NE555 for transient analysis, Sallen–Key for AC analysis, and a PIC programmer for non-simulatable detection) confirmed that all features work together. The source code is open and available at the GitHub repository in [11].

The internship gave practical experience that the coursework alone would not have provided. Reading the KiCad `pcbnew` API [2], debugging across two operating systems (Windows for KiCad and WSL2 Ubuntu for the supporting Python tools), and writing software for users who do not want to learn Git all shaped the final design in ways that are difficult to anticipate from the outside. These design choices have been recorded in this report and in the repository's design notes so that the next intern who picks up this work does not have to repeat them.

8.2 Limitations

The following limitations of the current version are documented in the same form in the repository's `KNOWN_ISSUES.md`, so that a future intern can pick any of them up directly.

- **Synchronous snapshot capture.** On a very large board (roughly 1000+ footprints), capturing a snapshot introduces a perceptible pause on save. Moving the snapshot write to a background thread would resolve this; the diff and timeline updates can stay on the UI thread.
- **Wire reroutes are not flagged.** The diff engine operates on the parsed semantic representation, so a wire reroute that preserves net membership is invisible. This is deliberate (a flood of visual-only diffs would drown out the real changes), but a `--verbose` option that includes layout deltas is a reasonable extension.
- **Single-machine assumption.** The plugin is designed for one user editing one project at a time. Cloud-synchronisation services such as OneDrive function as backing storage, but concurrent edits from two machines may produce conflicting snapshots that the plugin does not detect or merge.
- **Tag persistence depends on the diary folder.** Tags live inside the snapshot files. Deleting the diary folder loses tag history. There is no separate tag database.
- **No PCB-track-level diff.** The diff engine is component-centric. Changes to copper tracks, vias, and zones in the `.kicad_pcb` file are not currently surfaced. This is scoped as future work below.

8.3 Future Scope

Three concrete directions for extending this work have been scoped already and form natural continuations of the internship.

1. **Automatic netlist re-export on checkpoint.** At present, the user explicitly creates a simulation checkpoint, at which point the plugin invokes eSim's converter and NGSpice. The next iteration would automate netlist re-export so that whenever the schematic changes, a fresh netlist can be produced on demand. This is intended to integrate with related tooling under development by other FOSSEE contributors.
2. **Track-level diffing for the PCB.** The current diff engine is schematic-centric and identifies component-level changes. A natural extension is to extend the diff to copper tracks, vias, and zones in the `.kicad_pcb` file [6], giving a complete picture of both the logical and physical evolution of the design. The set-theoretic framework of Section 4.2 extends cleanly: the underlying domain $\mathcal{R}$ is enlarged to include track and via identifiers, and the property tuples grow to include geometric attributes.

3. **Subcircuit fingerprinting and semantic search.** A follow-on plugin, *Circuit-Sense*, has been scoped and proposed. It uses Weisfeiler–Leman graph hashing and VF2 subgraph isomorphism to identify recurring sub-circuits across a user's project history. The intended use case is to surface insights such as recurring filter patterns or repeated subsystem designs, making the design history searchable beyond a single project boundary. The fingerprint mechanism of Section 4.3 would generalise from schematic-level identity to subgraph-level identity.

These extensions are direct continuations of the work in this report and are intended to be pursued in subsequent FOSSEE engagements.

Bibliography

[1] KiCad Developers, *KiCad EDA — Official Documentation*, Version 9.0. Available at: https://docs.kicad.org/ [Accessed May 2026].

[2] KiCad Developers, *KiCad Python Scripting Reference — pcbnew Module*. Available at: https://docs.kicad.org/doxygen-python/ [Accessed May 2026].

[3] FOSSEE Project, IIT Bombay, *eSim: Free and Open Source EDA Tool for Circuit Design, Simulation, Analysis and PCB Design*. Available at: https://esim.fossee.in/ [Accessed May 2026].

[4] NGSpice Development Team, *NGSpice User's Manual*, Version 35. Available at: https://ngspice.sourceforge.io/docs.html [Accessed May 2026].

[5] wxPython Project, *wxPython Phoenix Documentation*. Available at: https://docs.wxpython.org/ [Accessed May 2026].

[6] KiCad Developers, *KiCad S-Expression File Format Specification*. Available at: https://dev-docs.kicad.org/en/file-formats/ [Accessed May 2026].

[7] FOSSEE, IIT Bombay, *Semester Long Internship Programme*. Available at: https://fossee.in/semester-internship/ [Accessed May 2026].

[8] S. Chacon and B. Straub, *Pro Git*, 2nd ed., Apress, 2014. Available at: https://git-scm.com/book

[9] P. P., "Embedding NGSpice UI in eSim and Testing Out ATTINY Microcontrollers," FOSSEE Semester Long Internship Report, 2023. Available at: https://static.fossee.in/fossee/reports-2023/semester_long_internship_2023/eSim/eSim_Pranav_P.pdf

[10] KiCad Developers, *KiCad Demos — Bundled Example Projects (simulation/sallen_key)*, KiCad source tree. Available at: https://gitlab.com/kicad/code/kicad/-/tree/master/demos/simulation/sallen_key [Accessed May 2026].

[11] S. Upadhyay, *KiCad Design Diary — Source Code and Design Documents*, Public GitHub Repository. Available at: https://github.com/Sia2005/kicad-design-diary

Appendix A

Daily Work Log

Day	Date	Phase	Work Description
Wed	18/02/2026	Orientation	Attended FOSSEE SLI Spring 2026 orientation session.
Thu	19/02/2026	Exploration	Explored internship tasks; studied KiCad plugin scope.
Fri	20/02/2026	Exploration	Studied prior intern work and FOSSEE eSim documentation.
Mon	23/02/2026	Exploration	Researched KiCad scripting API and ActionPlugin registration.
Wed	25/02/2026	Meeting	Attended Google Meet with mentor; discussed plugin direction.
Fri	27/02/2026	Proposal	Prepared plugin proposal document for idea pitch.
Mon	02/03/2026	Setup	Set up KiCad 9.0 on Windows 11 and WSL2 Ubuntu for Python tooling.
Thu	05/03/2026	Development	Implemented `ActionPlugin` registration and skeleton entry point.
Mon	09/03/2026	Development	Implemented snapshot capture from `pcbnew.GetBoard()`.
Thu	12/03/2026	Development	Added direct S-expression parsing for the `.kicad_sch` file.
Sat	14/03/2026	Meeting	Reviewed snapshot format with mentor; agreed on JSON layout.
Mon	16/03/2026	Diff Engine	Implemented semantic diff function; tested on hand-crafted snapshots.
Wed	18/03/2026	UI	Built initial wxPython `DiaryPanel` skeleton with ListCtrl.

Continued on next page

Day	Date	Phase	Work Description
Fri	20/03/2026	UI	Added colour coding for `SCH`, `PCB`, `SIM`, `ROLLB` rows.
Sat	21/03/2026	Meeting	Discussed rollback design; agreed on auto-save before rollback.
Mon	23/03/2026	Rollback	Implemented rollback with pre-rollback safety snapshot.
Wed	25/03/2026	UI	Added Tag Snapshot and Compare Snapshots buttons.
Sat	28/03/2026	Meeting	Reviewed core plugin behaviour; planned simulation integration.
Mon	30/03/2026	Simulation	Studied eSim KiCad-to-NGSpice converter invocation.
Thu	02/04/2026	Simulation	Implemented NGSpice batch-mode invocation via subprocess.
Sat	04/04/2026	Meeting	Debugged path-with-spaces issue with mentor; agreed on temp-dir workaround.
Mon	06/04/2026	Simulation	Implemented `.raw` file parsing and interactive waveform plot generation.
Wed	08/04/2026	Stale Detection	Implemented SHA-256 schematic hashing with whitespace normalisation.
Fri	10/04/2026	Stale Detection	Integrated stale flag into timeline panel and HTML export.
Mon	13/04/2026	Non-sim	Built regex pattern table for non-simulatable components.
Wed	15/04/2026	Non-sim	Added marker file (`_sim_warning_shown.json`) for popup deduplication.
Fri	17/04/2026	Error Translation	Implemented NGSpice error translator for common failure modes.
Mon	20/04/2026	HTML Report	Built HTML report generator with header, statistics, and timeline.
Wed	22/04/2026	HTML Report	Added Modification Frequency bar chart and embedded waveforms.
Mon	27/04/2026	Testing	End-to-end testing on NE555 astable multivibrator.
Wed	29/04/2026	Testing	End-to-end testing on Sallen–Key low-pass filter from KiCad demos.
Fri	01/05/2026	Testing	End-to-end testing on PIC microcontroller programmer (non-sim path).

Continued on next page

Day	Date	Phase	Work Description
Mon	04/05/2026	Refinement	Code cleanup, `KNOWN_ISSUES.md`, and `README.md` polish.
Tue	05/05/2026	PR	Submitted pull request to FOSSEE/eSim-KiCad-Plugin repository.
Wed	06/05/2026	Report	Drafted Chapters 1 to 3.
Thu	07/05/2026	Report	Drafted Chapter 4 (mathematical foundations) with formal definitions.
Fri	08/05/2026	Report	Drafted Chapter 5 (core plugin) with module diagram and tables.
Mon	11/05/2026	Report	Drafted Chapter 6 (simulation pipeline) and Chapter 7 (validation).
Tue	12/05/2026	Report	Final review and corrections; pseudocode alignment with implementation.
Wed	13/05/2026	Submission	Final report submitted.