Verifying Extract Method Refactoring in Rust

Matthew Britton
Australian National University
Canberra, Australia
matt.britton@anu.edu.au

Alex Potanin
Australian National University
Canberra, Australia
alex.potanin@anu.edu.au

Sasha Pak
Australian National University
Canberra, Australia
sasha.pak@anu.edu.au

Abstract

Refactoring is trustworthy only if semantics are preserved. In Rust, ownership and lifetimes make automatic extraction attractive yet risky: code that merely compiles can still change aliasing, lifetime structure, or observable effects. We present REM-V, an end-to-end pipeline that extracts, fixes, and verifies Extract-Method refactorings. Built atop the Rusty Extraction Maestro (REM), a toolchain for extraction and compiler-guided repairs via cargo check, REM-V performs lightweight, zero-annotation equivalence checking: P and P' are compiled to LLBC (CHARON) and functionalised (AENEAS), enabling automatic observational equivalence checks. Early results indicate feasibility and smooth IDE integration. A VSCode makes the Extract, Fix, Verify loop available to developers as VS Code plugin called "REM VSCode".

CCS Concepts: • **Software and its engineering** → *Integrated and visual development environments*; *Software maintenance tools*; *Formal software verification.*

Keywords: Automated Verification, Extract Method, Refactor as Repair, Rust

ACM Reference Format:
Matthew Britton, Alex Potanin, and Sasha Pak. 2025. Verifying Extract Method Refactoring in Rust. In *Companion Proceedings of the 2025 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3758316.3765486

1 Introduction & Motivation

Extract method refactoring improves readability and maintainability by lifting code fragments into named functions. Whilst very straightforward in garbage collected languages (think Java, Python, etc.), Rust's strict ownership and lifetime system complicates automated extraction. A misplaced borrow or incorrect lifetime can easily render code uncompilable, and in some cases, alter code semantics altogether.

SPLASH Companion '25, Singapore, Singapore

ACM ISBN 979-8-4007-2141-0/25/10
https://doi.org/10.1145/3758316.3765486

To address the compilation problem, the Rusty Extraction Maestro (REM) tool-chain was built, using compiler feedback to iteratively address problems introduced by the standard Extract Method mechanics. It achieved a 92% success rate across 5 popular crates [3, 4]. However, compilation alone cannot guarantee functional equivalence. In high assurance domains, a refactoring tool that *might* change behaviour is unacceptable.

1.1 REM in a Nutshell

REM is a compiler-guided Extract Method engine for Rust that turns a naïve hoist into a sequence of Rust-aware repairs: (i) it *reifies* non-local control flow (`return`/`break`/`continue`) into a small result `enum` so the caller can reconstruct the original flow; (ii) it infers, per free variable, whether to pass by value, shared borrow (`&T`), or unique borrow (`&mut T`); and (iii) it repairs lifetimes in a feedback loop, runs `cargo check` to read errors, iteratively fix signatures, then applies lifetime elision. The result is a compiling, readable extraction that preserves borrow discipline and control-flow.

2 Approach

Our goal is to *verify* that an Extract Method transformation preserves behaviour. We build on REM's existing success and add an automated, zero-annotation equivalence check.

Pipeline

Our goal is to refactor a selected region and then certify that the refactoring preserves behaviour at the call boundary.
Pipeline.

1. **Extract & Fix (REM):** Extracts a user selected area, repairs control-flow, passing modes, and lifetimes to produce a compiling program P' from P [4].
2. **Fix the boundary:** compare the original caller in P (inlined region) to the modified caller in P' using REM's repaired signature and result `enum`.
3. **CHARON (LLBC):** compile P and P' to an ownership-explicit IR with moves/borrows/drops made explicit [1, 2].
4. **AENEAS (functionalisation):** translate LLBC to a pure λ-calculus while preserving behaviour.
5. **Equivalence check (Coq):** We automatically discharge *observational equivalence* at the extracted call: for all inputs satisfying the same preconditions, P and P' return the same result and exhibit the same externally visible effects - all without user annotations.[1]

[1]Example output available at: https://github.com/RuleBrittonica/REMV-SPLASH25-Posters/tree/main/complete_verification_output

Equivalence obligation. Write $[\![f]\!]^P_\lambda(x) = (r, \tau)$ for the functionalised semantics (via Aeneas) of f in program P on input x, yielding a result r and an effect trace τ; similarly for P'. Let $\Phi(x)$ encode the call-site preconditions induced by the repaired signature and result enum, and let obs project the observable portion of a trace. The obligation we prove is:

$$\forall x.\ \Phi(x) \Rightarrow \Big([\![f]\!]^P_\lambda(x) = (r, \tau) \ \wedge \ [\![f]\!]^{P'}_\lambda(x) = (r', \tau')\Big) \ \Rightarrow \ \big(r = r' \ \wedge \ \mathrm{obs}(\tau) = \mathrm{obs}(\tau')\big). \quad (1)$$

In effect-free fragments τ and τ' are empty (or observationally equal), so the goal reduces to plain extensional equality of results: $\forall x.\ \Phi(x) \Rightarrow [\![f]\!]^P_\lambda(x) = [\![f]\!]^{P'}_\lambda(x)$.

3 Preliminary Results

We adapted ten cases from the *rust-analyzer* tests, each isolating one design feature. Table 1 reports the refactored signature, LOC before/after, and whether the check discharged automatically. Full cycles (extract / proof gen / verify) averaged ~2s, suggesting IDE-friendly latency. [2] Though not our focus, a full cycle (extract-fix-verify) completes in 2 s, suggesting real-world viability.

Table 1. Ten extraction sites adapted from the *rust-analyzer* test suite.

ID	Focus	`refactored sig`	LOC $P \to P'$	Equiv
0	break loop	`n: i32 → Option<i32>`	11→18	Y
1	break with value	`() → Option<i32>`	13→19	Y
2	comments in block expr	`() → i32`	9→10	Y
3	extract from nested	`() → i32`	9→12	Y
4	extract method from trait impl	`&self → i32`	11→16	Y
5	extract mut ref	`y: &mut Foo`	14→17	Y
6	extract return stmt	`() → u32`	5→8	Y
7	mut method call	`mut n: i32`	12→15	Y
8	no args if let else	`() → i32`	5→8	Y
9	try option with return	`() → Option<i32>`	12→16	Y

4 Future Work & Limitations

Current Limitations

Verification backend. "We currently target *safe* Rust; `unsafe` code is out of scope" [2]. Known AENEAS limits include: no nested loops; no function pointers or closures (ongoing); limited type parametricity ; no nested borrows in function signatures (ongoing); incomplete modelling of interior mutability (e.g. `Cell`/`RefCell`); and no concurrency (long-term) — even coarse-grained parallelism needs additional semantics.
Language/features coverage. End-to-end proofs are not yet guaranteed for code using complex generics, heavy macro expansion (incl. proc-macros), `async`/`await`, or intricate trait bounds. FFI, platform I/O, and panic/abort behaviour are only partially abstracted and may be treated as uninterpreted oracles.
What the proof says. We check observational equivalence at the extraction boundary under the current abstraction of side effects. If the program relies on global state, time, or non-determinism, additional modelling is required.

Future Work

Expanded Evaluation To ensure reliability, the extraction and verification engine must be tested against hundreds of different cases. We aim to pull the majority of these from major, ongoing rust projects, such as Deno.
Unsafe and FFI. Compose with tools that target `unsafe` (e.g. borrow-aware model checking or type-system-based frameworks) and introduce contracts for FFI boundaries to keep equivalence checks meaningful.
Concurrency. The extraction engine is able to work with concurrent code, however, to integrate verification we will start with coarse-grained parallelism (task spawning/joining) under a sequentially-consistent abstraction.
IDE integration Work is already underway to create a VS-Code extension that makes the functionality easily accessible to the developer.
Better diagnostics. When equivalence fails, surface a small, source-level counterexample trace at the call site (inputs, branch taken, differing outputs). Integrate this feedback into the editor so that developers can immediately adjust the extraction.
Scalable verification. Adopt slice-based translation by default with sound stubbing of unrelated items; caching LLBC and proof artefacts for incremental re-verification. This will allow for significantly shorter overall verification times.

References

[1] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024. Sound Borrow-Checking for Rust via Symbolic Semantics. *Proc. ACM Program. Lang.* 8, ICFP, Article 251 (Aug. 2024), 29 pages. doi:10.1145/3674640
[2] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (Aug. 2022), 31 pages. doi:10.1145/3547647
[3] Sewen Thy. 2023. Borrowing without Sorrowing: Implementing Extract Method Refactoring for Rust. (2023). https://verse-lab.github.io/papers/Sewen-Thy-Capstone.pdf
[4] Sewen Thy, Andreea Costea, Kiran Gopinathan, and Ilya Sergey. 2023. Adventure of a Lifetime: Extract Method Refactoring for Rust. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 245 (Oct. 2023), 28 pages. doi:10.1145/3622821

Received 2025-08-18; accepted 2025-08-24

[2] Cases are available at: https://github.com/RuleBrittonica/REMV-SPLASH25-Posters/tree/main/verification_examples