Applying Verified Z3 Proof Checking to Ladder Logic Verification of Railway Interlockings

Harry Bryant$^{1[0009-0008-9926-8678]}$, Anton Setzer$^{1[0000-0001-5322-6060]}$, Andrew Lawrence^{2}, and Monika Seisenberger$^{1[0000-0002-2226-386X]}$

^{1} Swansea University, SA1 8EN, Wales, UK
harry.bryant@swansea.ac.uk, a.g.setzer@swansea.ac.uk, and m.seisenberger@swansea.ac.uk
^{2} Siemens Mobility Limited (UK)
andrew.lawrence@siemens.com

Abstract. Railway systems are safety critical and demand the highest levels of assurance for their control software. Formal verification tools, used alongside conventional testing, are essential for ensuring compliance with stringent safety and regulatory standards. In this article, we present a solution with the goal to be part of a future verification toolchain involving Z3 SAT/SMT solving and to be certified at Safety Integrity Level 4 (SIL4). We demonstrate this via a verified and formally extracted SAT proof checker in the context of Ladder Logic verification. Our approach is tailored to the needs of our industrial partner, adaptable, and also extendable to include further SMT theories. Our proof checker currently works for Z3's full propositional proof output, including Tseitin transformations. A checker for Z3 proofs of CNF formulas has been formalised and verified in Rocq, with a certified OCaml implementation extracted from the proof, whilst the full extendable framework including RUP inferences and Tseitin transformation has been fully verified in Agda. Our approach enables formal reasoning about Z3 outputs in both theorem provers. Finally, we demonstrate the entire approach with a small case study, and provide results on the scalability on an industrial level.

Keywords: Railway Verification · Verification · Proof Checking · Coq · Rocq · Ladder Logic · Interlocking Systems · Z3 · SAT solving · SMT solving · RUP · Tseitin transformation.

1 Introduction

Railway systems are among the most safety-critical infrastructures, where software failures can have severe consequences. To ensure safety and regulatory compliance, railway control systems, particularly interlockings, must be validated against formalised safety requirements, such as those defined by Network Rail in the UK [1]. These are based on operational procedures, historical incidents, and formal design rules. Validation typically involves weeks of testing, and any failure requires costly redesign and retesting. To mitigate this, formal verification tools (see e.g. [2]–[4]) are increasingly used early in development. These tools automatically check interlocking designs against formal safety properties, helping to catch issues before physical testing. One such tool is the Ladder Logic

Verifier [5], which targets interlockings written in ladder logic [6], a graphical language defined by the IEC 61131 standard [7]. The approach proves the unreachability of unsafe states by demonstrating the unsatisfiability of the negated safety property. Many verification tools use Z3 [8], a leading SMT solver, but its internal complexity makes independent validation challenging. To support Safety Integrity Level 4 (SIL4) certification, we adopt a proof-checking approach (e.g. [9]) that verifies Z3's unsatisfiability proofs rather than the solver itself. This ensures interlockings proceed to industrial testing only after both verification and proof validation are complete (Fig. 1).

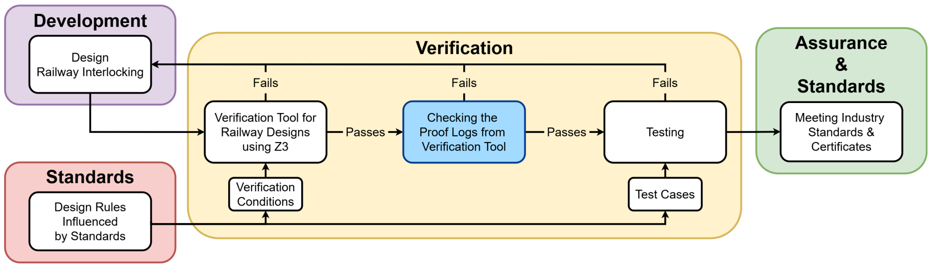

Development
Design Railway Interlocking
Verification
Fails
Fails
Fails
Verification Tool for Railway Designs using Z3
Passes
Checking the Proof Logs from Verification Tool
Passes
Testing
Assurance & Standards
Meeting Industry Standards & Certificates
Standards
Design Rules Influenced by Standards
Verification Conditions
Test Cases

Fig. 1: Proposed railway interlocking design methodology: The interlocking and the safety properties are translated into SMTLIB [10] for analysis with Z3. If successful, then our proof checker validates the proof log of unsatisfiability. Only if both are successful, then industrial testing takes place.

We present a formally verified proof checker for Z3's RUP (Reverse Unit Propagation) proof format, introduced in 2022 [11]. The RUP checker, developed entirely in the Rocq proof assistant [12] and extracted to OCaml [13]–[15], validates full propositional proofs, including Tseitin transformations[3]. However, here only the RUP steps are fully verified. We also introduce a prototype full checker for verifying both Tseitin and RUP steps as part of a broader, extensible toolchain in Agda. This enables independent validation of solver outputs and provides a certifiable link between SMT solving and formal proof checking. Our approach integrates with Rocq/Agda, verifying the actual program rather than just the algorithm. It is designed to be extensible and adaptable to future changes in Z3's proof format, ensuring long-term maintainability. Currently, it deals with general propositional formulas, not just CNFs [17]. Developed in collaboration with our industrial partner, who required the use of Z3 due to its certification status, the checker has been evaluated on industrial-scale interlocking systems. While our current focus is on discrete interlockings written in ladder logic, the methodology is generalisable to modern railway systems such as ERTMS and the verification of scheme plan data [4]. By independently validating solver outputs, our approach eliminates a key point of failure in automated verification pipelines. It contributes directly to certification efforts and strengthens safety assurance in railway software engineering.

[3] The Agda implementation of the RUP checker has been presented in [16], the formalisation in Rocq and the treatment of Tseitin is new.

Related Work. SAT/SMT competitions nowadays require not just answers but also independently checkable unsatisfiability proofs. This has driven extensive research into proof checking and reconstruction across using different proof formats and tools (see e.g. [18]–[29]).[4] The currently fastest verified SAT Checker is presented in [30] using LRAT, a modification of the DRAT format.

Recent work using cvc5 [31]–[33] provides a good framework for SMT proof checking and also introduces a modular proof architecture intended to support integration with in SMTCoq, Isabelle and Lean. Feng et al. [34] provide and approach to SMT proof checking SAT Modulo Monotonic Theories using MONOSAT. Finally, the Lambdapi proof checker project [35] aims at providing a platform to exchange proofs between different interactive proof assistants. While Z3 (RUP) outputs could be converted to other formats for checking[5], this adds overhead and reduces clarity - undesirable in industrial settings where simplicity and transparency are critical. This motivated our development of a checker tailored to Z3's native format.

Earlier work aimed at directly verifying SAT solvers (see, for instance, [36], [37] for formalisations in Rocq and Isabelle).Versat [38] is verified solver that includes CDCL, though their C implementation was only manually derived. In [39], a provably correct DPLL prover has been automatically extracted from a formal proof, and the extracted solver has been applied in Railway verification case studies.

Contributions. The main contribution of this article is a verified proof checker that supports SIL certification in railway verification (Fig. 1). It is extracted and applied to discrete interlocking systems, with support for full propositional logic via Tseitin transformation. This also serves as a proof-of-concept for extending the approach to additional Z3 theories.[6] The main results are:

1. Demonstration of an extensible proof checker for validating Z3's unsatisfiability proofs, supporting SIL compliance in railway verification. We present a tailored solution to fit the requirements[7] of our industrial partner.
2. Checker works for full Z3 propositional proofs, and is not restricted to formulas in CNF (e.g. DIMACS format); i.e., the checker can process Z3 proofs which include Tseitin steps for translating propositional formulas into CNF.
3. The checker itself is fully verified in Agda, and the RUP checker is verified in Rocq. We provide a tool chain from SMT solver to Agda/Rocq, i.e. a proof of correctness of our SMT proof checker, which also allows to integrate theorems in Z3 into Agda and Z3 theorems in CNF into Rocq.

[4] Interestingly we could not find an off the shelf solution to fit the format of our industrial needs. Therefore we decided for own development to meet the requirements by our partner, and be able to further extend it with the Z3 theories we need.

[5] We are aware that there is an internal RUP checker in Z3, and also the option to produce a (longer) resolution proof (Z3's old proof format).

[6] Z3 theories are essential for verifying advanced railway systems such as ERTMS and geographic data [4].

[7] Formats/tools used form part of an overarching certification framework, involving Z3.

4. Small case study[8] to demonstrate the entire tool qualification process.
5. A GitHub repository [40] containing the SAT checker code in Rocq, Agda, and OCaml, as well as our case study.

2 Z3 Proofs of Unsatisfiability

The Z3 SMT solver [8] is widely used to verify whether an interlocking satisfies a propositional safety property. It checks the satisfiability of the negated property $\neg\varphi$ combined with the interlocking model (see James et al. [41]). If unsatisfiable, the original property φ holds, and Z3 can generate a proof log [42] for independent checking. Otherwise, a counterexample is returned [43]–[46]. The Ladder Logic Verifier uses Z3 first in inductive verification [41], [47], and falls back to bounded model checking [48], [49] if needed. To illustrate, we model a simple interlocking with a passing loop (Fig. 2) [50], where a signal turns green only if the point is correctly set and the opposing signal is red. The safety property ensures that opposing signals are never green at the same time. We assert its negation, $(s0 \wedge s1) \vee (s2 \wedge s3)$, and Z3 returns `unsat`, confirming the design is safe. Z3's proof log [42] shows the logical steps leading to unsatisfiability (Fig. 3). The proof includes nine assumption steps, 18 Tseitin steps, and seven RUP steps. Assumption introduces disjunctions, Tseitin encodes formulas into CNF [17], [51]–[53], and RUP derives new clauses.

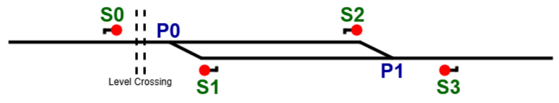


Fig. 2: Simple railway interlocking with a passing loop

```
tseitin(Not(And(Not(s1), p0)), Not(s1)) [] [Not(s1), Not(And(Not(s1), p0))]
tseitin(Not(And(Not(s1), p0)), p0) [] [p0, Not(And(Not(s1), p0))]
tseitin(s1, Not(p0), And(Not(s1), p0)) [] [s1, Not(p0), And(Not(s1), p0)]
assumption [] [Not(s0), And(Not(s1), p0)]
-- intermediate steps omitted --
rup [] [Not(And(Not(s2), Not(p1)))]
rup [] []
```

Fig. 3: Snippet of Z3 proof log of unsatisfiability of the railway interlocking

The Tseitin transformation introduces fresh variables to represent subformulas, preserving equisatisfiability while enabling efficient SAT solving. In Z3, however, these fresh variables are replaced directly by the original formulas, resulting in tautological clauses. Our prototype checker validates Tseitin steps by matching them against known tautological patterns including:

[8] Due to an NDA agreement in place we cannot demonstrate the process using a real world interlocking, but we will report on the scalability of the approach.

- `And`:
  - `tseitin(Not(And(`$a_1, ..., a_n$`)), `a_i`) [] [`a_i`, Not(And(`$a_1, ..., a_n$`))]`
  - `tseitin(Neg(`a_1`),..., Neg(`a_n`), And(`$a_1, ..., a_n$`)) []`
    `[Neg(`a_1`),..., Neg(`a_n`), And(`$a_1, ..., a_n$`)]`
- `Not`:
  - `tseitin(b, Not(b)) [] [b, Not(Not(Neg(b)))]`
- `Or`:
  - `tseitin(Neg(`a_i`), Or(`$a_1, ..., a_n$`)) [] [Neg(`a_i`), Or(`$a_1, ..., a_n$`)]`
  - `tseitin(`$a_1, ..., a_n$`, Not(Or(`$a_1, ..., a_n$`))) []`
    `[`$a_1, ..., a_n$`, Not(Or(`$a_1, ..., a_n$`))]`

We implemented a prototype checker for these Tseitin steps in Agda [54], a dependently typed language and proof assistant. The checker verifies that each Tseitin clause matches a valid tautological pattern. Correctness is formally proven by showing that all such clauses are tautologies, ensuring that the transformation preserves equisatisfiability [55]. Together with our formally verified RUP checker, these components form a complete SMT proof checker for propositional formulas. Future work will focus on translating the Tseitin checker to Rocq and integrating it with the RUP checker for industrial deployment. Using the correctness proofs of both the Tseitin and RUP checkers, we establish the following key result: if the proof checker returns true, then the assumptions of the Z3 proof entail its conclusions (Fig. 4). Furthermore, if in addition the assumptions contain the empty clause, the proof confirms unsatisfiability (Fig. 5).

```
    correctnessZ3ProofCheck : (p : ZProof)
                            → atom (ZProofCheck p)
                            → EntailsListZCl (ZProof2Assumption p
                              (ZProof2ConclusionOpt p)
```

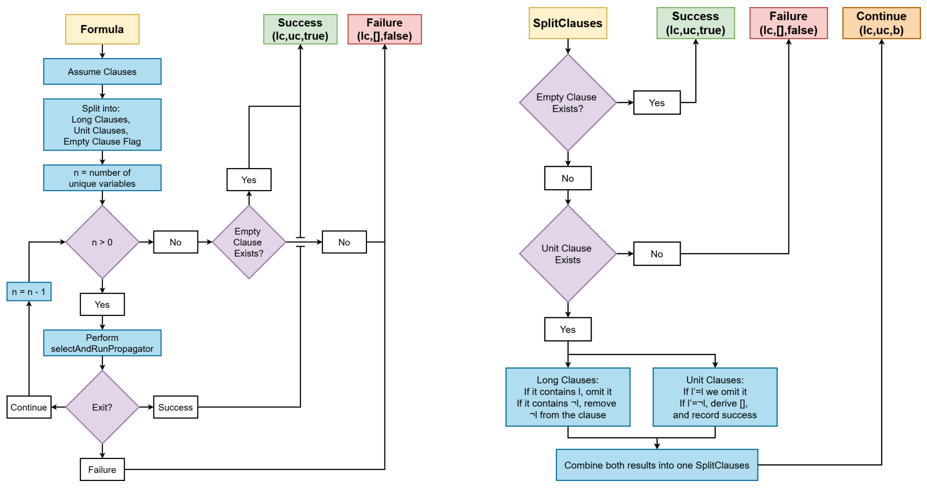
Fig. 4: If the checker is true, then the proof's assumptions entail its conclusions

```
correctenessZ3ProofCheckUnsat : (p : ZProof)
                                → atom (ZProofCheckUnsat p)
                                → UnSat (ZProof2Assumption p)
```

Fig. 5: Proof of checker returning [] confirms the assumptions are unsatisfiable

3 Three Level Approach for a Verified RUP Checker

The basis of the new Z3 proof log format is Reverse Unit Propagation (RUP) [56]–[58]. In a RUP proof each inference, of the form `rup [a]`, indicates that the clause `a` has been derived and validated by showing that its negation and a formula `f` leads to a contradiction via unit propagation. Ultimately, the proof concludes with `rup []`, signifying that falsity has been reached directly, without needing to derive any further clauses (success) or no further steps are possible (failure). Clauses can be of length ≥ 2, which we call *long clauses*, unit clauses, which are clauses of length 1, or the empty clause. In RUP, a formula is a conjunction of clauses written in CNF.


Formula
Assume Clauses
Split into: Long Clauses, Unit Clauses, Empty Clause Flag
n = number of unique variables
n > 0
No
Empty Clause Exists?
Yes
No
Success (lc,uc,true)
Failure (lc,[],false)
n = n - 1
Yes
Perform selectAndRunPropagator
Exit?
Continue
Success
Failure
SplitClauses
Success (lc,uc,true)
Failure (lc,[],false)
Continue (lc,uc,b)
Empty Clause Exists?
Yes
No
Unit Clause Exists
No
Yes
Long Clauses: If it contains l, omit it If it contains ¬l, remove ¬l from the clause
Unit Clauses: If l'=l we omit it If l'=¬l, derive [], and record success
Combine both results into one SplitClauses

(a) Unit Propagation Checker Process (b) selectAndRunPropagator Process

Fig. 6: Overview of the RUP checker processes

Termination is guaranteed by structurally reducing variables at each step, satisfying Rocq's requirements. The implementation follows a loop that applies one-step unit propagation (Fig. 6a): clauses containing l are removed, and those with $\neg l$ are reduced to $c' := c \setminus \neg l$. Unit clauses are handled similarly - if $l' = l$, it is omitted; if $l' = \neg l$, the empty clause is derived; otherwise, it is retained (Fig. 6b). The checker and its formal correctness proofs were developed in Rocq and are available on GitHub [40]. We extracted the implementation to OCaml using Rocq's built-in mechanism [13]–[15], and built a parser for Z3 proof logs. To prove correctness, we formalised the `TreeProof` datatype in Rocq, representing unit resolution proofs.[9]

```
Inductive TreeProof : Type := | ass : nat → TreeProof
                              | ures : TreeProof → TreeProof → TreeProof.
```

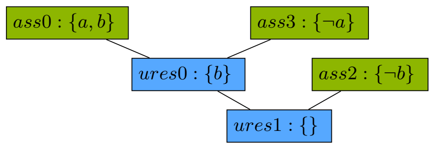


Fig. 7: Example of Tree proof deriving a contradiction.

RUP is proven via unit propagation, which employs a series of unit resolution steps. Here `(ass n)` denotes the assumption rule, deriving the `n`th assumption. The assumptions are the original set of clauses prior to unit propagation. For each unit resolution cut, we create `(ures t1 t2)` which denotes a `TreeProof` of unit resolution from `TreeProofs` `t1` and `t2`. For example, we derive that the formula $\{a, b\}$, $\{\neg a, b\}$ and $\{\neg b\}$ is unsatisfiable. The RUP format first derives $\{a\}$, by adding $\{\neg a\}$ to the assumptions and deriving falsity using unit resolution

[9] The term "tree proof" distinguishes this format from SMT and Rocq proofs.

(Fig. 7). An extended version of our Rocq code generates Tree Proofs, offering additional confidence in the checker by providing a trace of the resolution cuts.

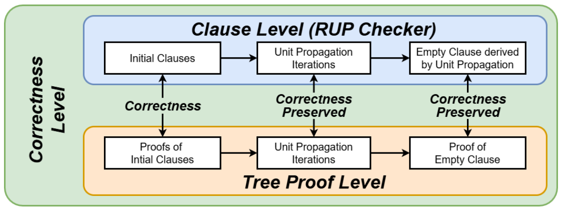


Fig. 8: The three level architecture of the RUP proof checker

To prove soundness, we show that if the RUP checker returns `true`, then there exists a valid `TreeProof` of the empty clause from a RUP step `c'`. We implemented a unit resolution checker operating on long clauses and unit clauses. Each clause-level operation is mirrored at the `TreeProof` level to produce a matching resolution trace. We proved these operations preserve correctness with respect to the original clauses (Fig. 8). A `TreeProof t` is correct if it derives a clause `c`, from assumptions `al`, that matches the result on the clause level:

```
Definition CorrectProof (al : Assumption)(c : Clause)(t : TreeProof) : Prop :=
  correctConclusion al t = Some c.
```

A `TreeProof` is valid if every use of `(ass n)` refers to a valid assumption `(n < length al)`, and each subproof of the form `(ures t1 t2)` where `t2` is a unit clause. This ensures that `c'` entails falsity in all models. Since `c'` is defined as `negate_clause c ++ a`, its unsatisfiability implies the original assumptions `a` entail the RUP clause `c`. Therefore, if `[]` can be derived from `a`, then `a` is unsatisfiable. By the definition of entailment [59], any model satisfying the assumptions must also satisfy the conclusion. If the assumptions are true in a `model`, so is the clause. As in the Agda prototype, we proved that if the RUP checker returns true, the empty clause was derived, and the assumptions entail it. This implies the correctness of RUP proofs and therefore of Z3 proofs of formulas in CNF:

```
Lemma RUP_Checker_correct : forall (a : Assumption) (c : Clause),
  RUP_Checker a c = true → entails a c.

Lemma RupProofcheckerUnSat : forall (pl : RupProof),
  rupProofCheckerUnsat pl = true → UnSatFor (rupProof2Assumptions pl).
```

4 Applying the Z3 SAT Proof Checker

The checker consists of two components. The first is the reverse unit propagation procedure extracted from Rocq, extended to support `assumption` and `Tseitin` steps for SAT proof logs. Full checking of `Tseitin` steps will be added in future work. The second is a parser that reads Z3 proof logs line by line, converting each entry into the datatypes written in Rocq. The checker processes the proof sequentially, confirming validity if all steps succeed, or reporting the first invalid step encountered. The proof log from the example interlocking (Fig. 3) was successfully validated. To evaluate scalability, we tested the checker on an industrial interlocking with 75,000 propositional variables and 12,000 ladder logic rungs.

Two proof logs were generated using Inductive Verification (IV) and Bounded Model Checking (BMC). The IV logs contained around 29,000 steps, while BMC logs exceeded 500,000, reflecting BMC's higher computational cost. All logs were successfully validated (Table 1), with IV completing significantly quicker[10].

Approach	Proof Logs	Validity	Average Runtime
BMC	13	All Valid	7 hours, 49 minutes, 54 seconds
IV	6	All Valid	2 minutes, 50 seconds

Table 1: RUP proof checker runtimes on the proof logs from running the Ladder Logic Verifier on industrial interlockings

5 Conclusion

This work introduces a verified Z3 SAT proof checker to strengthen trust in railway verification tools such as the Ladder Logic Verifier. A key component is a formally verified Tseitin transformation, implemented in Agda, which converts arbitrary formulas into CNF. Each transformation step is proven to be a tautology, ensuring logical equivalence with the original formula. This framework is designed for extensibility, allowing additional SMT proof rules to be incorporated and verified within the same formal setting. The RUP checker, developed in Rocq, is an extracted verified program that validates Z3-generated RUP proofs. A key strength of our approach is that the checker operates directly on proof logs, avoiding inefficient translations into formats like resolution proofs. While such conversions are possible (e.g., using legacy Z3 formats), they are impractical for large-scale systems. Our method eliminates this overhead by using unit resolution trees as a formal foundation. Though not required during checking, these trees are useful for small examples and debugging. The extracted OCaml implementation supports full Z3 SAT traces and is designed for extensibility, enabling future support for theory-specific rules [42]. Developed with our industrial partner, the checker integrates into safety-critical toolchains and supports real-world SIL certification efforts. Our long-term goal is to automate and to verify the encoding of interlocking system designs into SMT-LIB [10], enabling broad safety property verification through a formally verified, extensible infrastructure.

Further steps will include: (1) Translating the Tseitin checker written in Agda into Rocq and integrating it with the current extracted checker. (2) Extracting the checker to C using CertiCoq [60], followed by performance testing and scalability improvements. (3) Formalising the proof in Rocq's safe core, leveraging its infrastructure to ensure the core logic is verified within a minimal, trusted kernel. (4) Exploring certified program extraction, evaluating existing frameworks for full certification and identifying limitations. (5) Extending the checker to full SMT by integrating provably correct rules like Farkas, and adding support for more complex railway verification tasks. (6) Applying the checker in our toolchain for verifying geographic scheme data [4], and developing larger case studies to evaluate scalability and robustness.

[10] Tests were run on a machine with 128 64-core processors at 2194.443 MHz.

References

[1] Network Rail. *Signalling Principles Handbook: Interlocking Principles (Former Railway Group Standard GK/RT0060) - NR/L2/SIG/30009/GKRT0060*. Available at `https://www.rssb.co.uk/standards-catalogue/CatalogueItem/GKRT0060-Iss-4`. Dec. 2020.

[2] Anne Haxthausen and Jan Peleska. "Efficient Development and Verification of Safe Railway Control Software". In: *Railways: Types, Design and Safety Issues*. Ed. by Cacilie Reinhardt and Klaus Shroeder. Nova Science Publishers, 2013, pp. 127–148. ISBN: 978-1-62417-139-0.

[3] Phillip James, Andy Lawrence, Faron Moller, et al. "Verification of Solid State Interlocking Programs". In: *Software Engineering and Formal Methods*. Ed. by Steve Counsell and Manuel Núñez. Cham: Springer International Publishing, 2014, pp. 253–268. ISBN: 978-3-319-05032-4.

[4] Madhusree Banerjee, Victor Cai, Sunitha Lakshmanappa, et al. "A Tool-Chain for the Verification of Geographic Scheme Data". In: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*. Ed. by Birgit Milius, Simon Collart-Dutilleul, and Thierry Lecomte. Cham: Springer Nature Switzerland, 2023, pp. 211–224. ISBN: 978-3-031-43366-5. DOI: `10.1007/978-3-031-43366-5_13`.

[5] Simon Chadwick, Phillip James, Markus Roggenbach, et al. "Formal Methods for Industrial Interlocking Verification". In: *2018 International Conference on Intelligent Rail Transportation (ICIRT)*. 2018, pp. 1–5. DOI: `10.1109/ICIRT.2018.8641579`.

[6] Nicolas DeGuglielmo, Saurav Basnet, and Douglas Dow. "Introduce Ladder Logic and Programmable Logic Controller (PLC)". In: *Proceedings of the 2020 IEEE Applied Systems and Engineering Environment for Emerging Nations Conference (ASEE-NE)*. Oct. 2020, pp. 1–5. DOI: `10.1109/ASEENE51624.2020.9292646`.

[7] International Electrotechnical Commission. *IEC 61131-3:2013 - Programmable controllers – Part 3: Programming languages*. 3.0. Accessed: 2025-06-10. International Electrotechnical Commission, Feb. 2013. URL: `https://webstore.iec.ch/en/publication/4552`.

[8] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992. URL: `https://dl.acm.org/doi/abs/10.5555/1792734.1792766`.

[9] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, et al. "Efficient Certified RAT Verification". In: *Automated Deduction – CADE 26*. Ed. by Leonardo de Moura. Cham: Springer International Publishing, 2017, pp. 220–236. ISBN: 978-3-319-63046-5. DOI: `10.1007/978-3-319-63046-5_14`.

[10] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.0.* `https://smt-lib.org`. 2010.

[11] Z3Prover contributors. *Z3 Theorem Prover, Version 4.11.2.* `https://github.com/Z3Prover/z3/releases/tag/z3-4.11.2`. Introduced RUP-based proof format. Accessed June 2025. Sept. 2022.

[12] Rocq Development Team. *Rocq.* `https://rocq-prover.org/`. Accessed: 2025-05-09. 2025.

[13] Pierre Letouzey. "Extraction in Coq, an Overview". In: *Logic and Theory of Algorithms: 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008 Proceedings 4.* Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. June 2008, pp. 359–369. ISBN: 978-3-540-69407-6. DOI: `10.1007/978-3-540-69407-6_39`.

[14] Jean-Christophe Filliâtre and Pierre Letouzey. *Extraction of programs in OCaml and Haskell.* `https://coq.inria.fr/doc/V8.11.1/refman/addendum/extraction.html`. From Coq 8.11.1 documentation, `https://coq.inria.fr/doc/V8.11.1/refman/addendum/extraction.html`. 2020. URL: `https://coq.inria.fr/doc/V8.11.1/refman/addendum/extraction.html`.

[15] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. "Verified Extraction from Coq to OCaml". In: *Proceedings of the ACM on Programming Languages* 8.PLDI (June 2024), pp. 52–75. DOI: `https://doi.org/10.1145/3656379`. URL: `https://doi.org/10.1145/3656379`.

[16] Harry Bryant, Andrew Lawrence, Monika Seisenberger, et al. *Verifying Z3 RUP proofs with the interactive theorem provers Coq/Rocq and Agda.* To appear in proceedings of Types 2025. June 2025.

[17] Alan G. Hamilton. *Logic for Mathematicians.* eng. Rev. ed. Cambridge: Cambridge University Press, 1988. ISBN: 0521368650.

[18] Sascha Böhme and Tjark Weber. "Fast LCF-Style Proof Reconstruction for Z3". In: *Interactive Theorem Proving.* Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 179–194. ISBN: 978-3-642-14052-5. DOI: `https://doi.org/10.1007/978-3-642-14052-5_14`.

[19] Sascha Böhme. *Proof Reconstruction for Z3 in Isabelle/HOL.* Workshop on Proof Exchange for Theorem Proving (PxTP). 2009. URL: `https://www21.in.tum.de/~boehmes/proofrec.pdf`.

[20] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. "Inprocessing Rules". In: *Automated Reasoning.* Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 355–370. ISBN: 978-3-642-31365-3. DOI: `https://doi.org/10.1007/978-3-642-31365-3_28`.

[21] Nathan Wetzler, Marijn Heule, and Warren Hunt. "DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs". In: *International Conference on Theory and Applications of Satisfiability Testing.* July 2014, pp. 422–429. ISBN: 978-3-319-09283-6. DOI: `10.1007/978-3-319-09284-3_31`.

[22] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. "Mechanical verification of SAT refutations with extended resolution". In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 229–244. ISBN: 978-3-642-39634-2. DOI: `https://doi.org/10.1007/978-3-642-39634-2_18`. URL: `https://doi.org/10.1007/978-3-642-39634-2_18`.

[23] Mathias Fleury and Hans-Jörg Schurr. "Reconstructing veriT proofs in Isabelle/HOL". In: *Electronic Proceedings in Theoretical Computer Science* 301 (Aug. 2019), pp. 36–50. ISSN: 2075-2180. DOI: `10.4204/eptcs.301.6`.

[24] Adrián Rebola-Pardo. "Even Shorter Proofs Without New Variables". In: *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Ed. by Meena Mahajan and Friedrich Slivovsky. Vol. 271. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 22:1–22:20. ISBN: 978-3-95977-286-0. DOI: `10.4230/LIPIcs.SAT.2023.22`. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2023.22`.

[25] CMU Transparency Group. *verified_rup: A Verified DRUP and DRAT Proof Checker*. `https://github.com/cmu-transparency/verified_rup`. Accessed: 2025-06-26. 2023.

[26] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. "Efficient Certified Resolution Proof Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 118–135. ISBN: 978-3-662-54577-5.

[27] SMTCoq Developers. *SMTCoq: Communication between Coq and SAT/SMT solvers*. `https://smtcoq.github.io/`. Accessed: 2025-06-26. 2025.

[28] Michael Armand, Germain Faure, Benjamin Grégoire, et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 135–150. ISBN: 978-3-642-25379-9.

[29] Mathias Fleury and Peter Lammich. "A More Pragmatic CDCL for IsaSAT and Targetting LLVM (Short Paper)". In: *Automated Deduction – CADE 29*. Ed. by Brigitte Pientka and Cesare Tinelli. Cham: Springer Nature Switzerland, 2023, pp. 207–219. ISBN: 978-3-031-38499-8.

[30] Peter Lammich. "Fast and Verified UNSAT Certificate Checking". In: *Automated Reasoning*. Ed. by Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt. Cham: Springer Nature Switzerland, 2024, pp. 439–457. ISBN: 978-3-031-63498-7.

[31] Haniel Barbosa, Clark Barrett, Martin Brain, et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore

Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9.

[32] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, et al. "Flexible Proof Production in an Industrial-Strength SMT Solver". In: *Automated Reasoning.* Ed. by Jasmin Blanchette, Laura Kovács, and Dirk Pattinson. Cham: Springer International Publishing, 2022, pp. 15–35. ISBN: 978-3-031-10769-6.

[33] Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, et al. "Better SMT Proofs for Easier Reconstruction". In: *AITP 2019 - 4th Conference on Artificial Intelligence and Theorem Proving.* Obergurgl, Austria, Apr. 2019. URL: `https://hal.science/hal-02381819`.

[34] Nick Feng, Alan J. Hu, Sam Bayless, et al. *DRAT Proofs of Unsatisfiability for SAT Modulo Monotonic Theories.* 2024. arXiv: `2401.10703 [cs.LO]`. URL: `https://arxiv.org/abs/2401.10703`.

[35] Alessio Coltellacci, Gilles Dowek, and Stephan Merz. "Reconstruction of SMT proofs with Lambdapi". In: *CEUR Workshop Proceedings.* Ed. by Giles Reger and Yoni Zohar. Vol. 3725. Montréal, Canada, July 2024, pp. 13–23. URL: `https://inria.hal.science/hal-04861898`.

[36] S. Lescuyer and S. Conchon. *A Reflexive Formalization of a SAT Solver in Coq.* In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar: 21st International Conference on Theorem Proving in Higher Order Logics. Proceedings of TPHOL08. Technical Report (2008-1-Ait Mohamed), pages 65 - 76. Available from `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=971acf742e8a2d29a56cf46343320966b18fff86`. 2008.

[37] F. Marić and P. Janičić. "Formal Correctness Proof for DPLL Procedure". In: *Informatica* 21.1 (2010), pp. 57–78.

[38] Duckki Oe, Aaron Stump, Corey Oliver, et al. "versat: A Verified Modern SAT Solver". In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by Viktor Kuncak and Andrey Rybalchenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 363–378. ISBN: 978-3-642-27940-9.

[39] Ulrich Berger, Andrew Lawrence, Fredrik Nordvall Forsberg, et al. "Extracting verified decision procedures: DPLL and Resolution". In: *Logical Methods in Computer Science* Volume 11, Issue 1 (Mar. 2015). ISSN: 1860-5974. DOI: `10.2168/lmcs-11(1:6)2015`. URL: `http://dx.doi.org/10.2168/LMCS-11(1:6)2015`.

[40] Harry Bryant, Andrew Lawrence, Monika Seisenberger, et al. *Verification of Z3 RUP Proofs in Coq-Rocq and Agda.* `https://github.com/HarryBryant99/Verification-of-Z3-RUP-Proofs-in-Coq-Rocq-and-Agda`. Accessed: 2025-06-10. 2025.

[41] Phillip James, Andy Lawrence, Faron Moller, et al. *Verification of Solid State Interlocking Programs.* Vol. 8368. Springer-Verlag London, Mar. 2014, pp. 253–268.

[42] Z3 Development Team. *Inference logs and proofs.* `https://microsoft.github.io/z3guide/programming/Proof%20Logs/`. Retrieved 16 June

2025 2025. URL: https://microsoft.github.io/z3guide/programming/Proof%20Logs/.

[43] Sahel Alouneh, Sa'Ed Abed, Mohammad Alshayeji, et al. "A comprehensive study and analysis on SAT-solvers: advances, usages and achievements". In: *Artificial Intelligence Review* 52 (Dec. 2019), pp. 2575–2601. DOI: 10.1007/s10462-018-9628-0.

[44] Markus Roggenbach, Antonio Cerone, Bernd-Holger Schlingloff, et al. *Formal Methods for Software Engineering Languages, Methods, Application Domains.* 1st ed. 1862-4499. Springer International Publishing, 2021, pp. 192–207.

[45] Julien Murzi and Lionel Shapiro. "Validity and Truth-Preservation". In: *Unifying the Philosophy of Truth.* Ed. by Theodora Achourioti, Henri Galinon, and José Martinez. Springer, 2015, pp. 431–459.

[46] H. Paul Williams. "The Satisfiability Problem and Its Extensions". In: *Logic and Integer Programming.* Vol. 130. International Series in Operations Research & Management Science. Boston, MA: Springer, 2009, pp. 105–144. DOI: 10.1007/978-0-387-92280-5_4.

[47] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Formal Methods in Computer-Aided Design (FMCAD 2000).* Vol. 1954. Lecture Notes in Computer Science. Springer, 2000, pp. 127–144. DOI: 10.1007/3-540-40922-X_8. URL: https://link.springer.com/chapter/10.1007/3-540-40922-X_8.

[48] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, Jan. 2001. ISBN: 978-0-262-03270-4.

[49] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008. ISBN: 026202649X.

[50] Wikipedia contributors. *Romney Sands railway station.* https://en.wikipedia.org/wiki/Romney_Sands_railway_station. Accessed: 2025-06-26. 2024.

[51] Grigori S. Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970.* Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28. URL: https://doi.org/10.1007/978-3-642-81955-1_28.

[52] Marius Minea. *Conjunctive Normal Form: Tseitin Transform.* H250: Honors Colloquium - Introduction to Computation. 2024. URL: https://people.cs.umass.edu/~marius/class/h250/lec2.pdf.

[53] Elias Kuiter, Sebastian Krieter, Chico Sundermann, et al. "Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023, pp. 1–13. ISBN: 9781450394758. DOI:

10.1145/3551349.3556938. URL: https://doi.org/10.1145/3551349.3556938.

[54] The Agda Team. *What is Agda?* https://agda.readthedocs.io/en/latest/getting-started/what-is-agda.html. Accessed: 2025-06-24. 2025.

[55] Markus Krötzsch. *Description Logic Rules.* IOS Press, 2010. ISBN: 978-1-61499-342-1.

[56] Allen van Gelder. *Verifying RUP Proofs of Propositional Unsatisfiability: Have Your Cake and Eat It Too.* In Proceedings of 10th International Symposium on Artificial Intelligence and Mathematics (ISAIM'08). 2008. URL: https://users.soe.ucsc.edu/~avg/ProofChecker/Documents/proofs-isaim08-long.pdf.

[57] Allen van Gelder. *Verifying RUP Proofs of Propositional Unsatisfiability.* Slides of a talk given at ISAIM'08. 2008. URL: https://users.soe.ucsc.edu/~avg/ProofChecker/Documents/proofs-isaim08-trans.pdf.

[58] Eugene Goldberg and Yakov Novikov. "Verification of Proofs of Unsatisfiability for CNF Formulas". In: *2003 Design, Automation and Test in Europe Conference and Exhibition.* Mar. 2003, pp. 886–891. DOI: 10.1109/DATE.2003.1253718.

[59] Pieter A. M. Seuren. "Logic and entailment". In: *The Logic of Language: Language From Within Volume II.* Oxford University Press, Oct. 2009, 85=87. ISBN: 9780199559480. DOI: 10.1093/acprof:oso/9780199559480.003.0001. URL: https://doi.org/10.1093/acprof:oso/9780199559480.003.0001.

[60] Andrew Appel, Yannick Forster, Joomy Korkut, et al. *CertiCoq.* https://certicoq.org/. Retrieved 16 June 2025 2025. URL: https://certicoq.org/.