Delft University of Technology
Master of Science Thesis in Embedded Systems

Verification of a DPLL Transition System in Rocq

Julia Dijkstra

Verification of a DPLL Transition System in Rocq

Master of Science Thesis in Embedded Systems

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Julia Dijkstra
j.dijkstra-9.tudelft.nl
juliadijkstra97@gmail.com

Author
Julia Dijkstra (j.dijkstra-9.tudelft.nl)
(juliadijkstra97@gmail.com)
Title
Verification of a DPLL Transition System in Rocq
MSc Presentation Date
June 19, 2026

Graduation Committee

Benedikt Ahrens	Delft University of Technology
Sebastijan Dumančić	Delft University of Technology

Abstract

This thesis presents a formal verification of an abstract transition-system presentation of the Davis-Putnam-Logemann-Loveland (DPLL) procedure in the Rocq proof assistant. Following Nieuwenhuis et al., SAT solving is modeled as a set of rule-based transitions between states rather than as a concrete algorithm. The thesis formalizes the syntax and semantics of propositional formulas, defines the classical and base DPLL transition systems, and proves their key metatheoretic properties. In particular, we establish correctness and completeness with respect to satisfiability, and we prove termination by showing that the transition relation is well-founded. The formalization extends the original abstract system by also including the pure literal rule. Building on the verified transition system, the thesis introduces an abstract notion of strategy and derives a terminating solver from any strategy satisfying suitable conditions. A concrete strategy is then implemented in Rocq using Equations and shown to satisfy the strategy specification. Together, these results provide a machine-checked and trustworthy core for SAT solving in Rocq and a basis for future extensions toward more advanced systems such as CDCL and DPLL(T).

Keywords and phrases DPLL, satisfiability, transition system, Rocq, verification

Contents

1 Introduction

The Satisfiability (SAT) problem asks whether a Boolean formula admits an assignment of truth values to its variables that makes the formula true. Despite this simple formulation, SAT is a central problem in both theoretical computer science and practical computing. It was the first problem shown to be NP-complete [10], and its generality makes it a natural target for encoding a wide range of combinatorial problems. As a result, improvements in SAT solving can often be translated into improvements for many other computational tasks.

SAT solvers are algorithms designed to decide this problem. Through sophisticated heuristics and search algorithms, they navigate enormous search spaces and make even very large SAT instances tractable [14]. Beyond optimization and planning tasks, SAT solvers have become critical tools for verifying large mathematical proofs that are infeasible to check by hand. Notable examples include the Boolean Pythagorean Triples problem [8], which showed that the integers from 1 to 7824 admit a two-coloring with no monochromatic Pythagorean triple, whereas this is impossible for the integers from 1 to 7825, and resulted in a 200-terabyte proof. Another example is Schur Number Five [7], where SAT solving was used to determine the fifth Schur number, that is, the largest n for which the integers from 1 to n can be colored with five colors without creating a monochromatic solution to $x + y = z$, producing a proof of roughly 2 petabytes.

These applications make it important to ensure the correctness of SAT solvers themselves since even small bugs could compromise the validity of the results they produce [5]. Interactive theorem provers (ITPs), such as Rocq [2] or Isabelle [17], provide a way to formally verify such systems and obtain strong guarantees about their behavior.

From a broader perspective, formally verified reasoning tools can also have societal value. SAT solvers and related software are used in settings where reliability matters, such as hardware verification, software verification, and safety-critical system design. In such contexts, increasing trust in the underlying software helps reduce the risk that incorrect results go unnoticed and propagate into downstream systems. A verified solver core also reduces dependence on separate external verification or proof-checking software to establish trust in the basic solving procedure, since its key correctness properties are already justified inside a trusted proof assistant. Although the present thesis works at an abstract level rather than at the level of an industrial solver, it contributes to this broader goal by providing a machine-checked and trustworthy core for SAT solving in Rocq.

The central objective of this thesis is to formalize and verify, in Rocq, the abstract transition-system presentation of SAT solving introduced by Nieuwenhuis et al. [16]. This presentation captures the Davis–Putnam–Logemann–Loveland (DPLL) procedure not as an executable algorithm, but as a collection of transitions between solver states. The main contribution of this thesis is to formalize these transition rules in Rocq and establish their key metatheoretic properties, namely correctness, completeness, and termination. This objective is motivated by two considerations. First, the transition-system view gives a compact and transparent account of SAT solving, which makes it particularly suitable for formal reasoning. Second, it provides a natural starting point for later extensions toward more expressive frameworks such as DPLL(T), and therefore a useful basis for future verification efforts.

Closely related work has already been carried out in Isabelle/HOL. In particular, Blanchette et al. [4] formalize abstract DPLL and CDCL calculi based on Nieuwenhuis et al. and refine them toward executable solvers. The contribution of the present thesis is therefore not to introduce the transition-system approach itself, but to establish it in Rocq. This is worthwhile both because Rocq provides a different logical setting, based on dependent

type theory, and because it makes this style of SAT-solver verification available in Rocq.

From this perspective, the aim of the thesis is not to verify a highly optimized modern SAT solver directly, but rather to verify the correctness of an abstract model of SAT solving from which concrete solving procedures can later be derived. In doing so, the thesis connects the abstract theory of DPLL with a machine-checked formalization in Rocq, and thereby provides a small but trustworthy verified core on which further extensions can build.

The thesis is organized as follows. Section 2 introduces the Rocq proof assistant and the aspects of its workflow that are relevant for this formalization. Section 3 presents the necessary background on propositional logic and satisfiability. Section 4 then introduces the transition system of Nieuwenhuis et al. [16] and its formalization, including proofs of correctness, completeness, and termination. Building on this, Section 5 shows how solving procedures can be derived from the transition system using strategies and presents a concrete implementation of such a strategy. Section 6 discusses related work and limitations of the formalization, and Section 7 concludes the thesis and discusses directions for future research.

The main contributions of this thesis are as follows.

- A full bi-implication for satisfiability is proved. Compared to Nieuwenhuis et al. [16], the thesis shows not only that derivations to a final non-fail state give a model, but also that, conversely, every satisfiable formula gives rise to such a derivation. For this converse direction, a new normalization procedure is introduced.
- The pure literal rule is included in the transition system. This rule is part of the original DPLL procedure, but Nieuwenhuis et al. [16] leave it out of their transition system. Adding it requires a strengthened entailment property, see Definition 39 and Lemma 40.
- An abstract notion of strategy is introduced, which makes it possible to derive verified solvers from the transition system. A concrete implementation of such a strategy is also given, and is extracted and tested on some small SAT instances.
- The whole development is formalized in Rocq.

2 The Rocq Prover

Rocq is an interactive theorem prover based on dependent type theory. More precisely, its logical foundation is the Calculus of Inductive Constructions, in which propositions are represented as types and proofs as terms inhabiting those types. This is an instance of the Curry–Howard correspondence, which underlies Rocq's treatment of proving and programming within a single formal language [3, 12].

At the user level, Rocq offers several different languages that play distinct roles. The first is *Gallina*, the specification language of Rocq. Gallina is used to define functions, predicates, inductive types, and theorem statements, and more generally to write the mathematical objects that make up a formalization [3]. The second is the vernacular command language, which provides commands for declaring definitions, starting proofs, importing libraries, and structuring developments. Finally, proofs are often constructed interactively using tactics, and one of the main tactic languages provided by Rocq is *Ltac*. Ltac scripts are not themselves proofs. Rather, they are programs that guide proof search and transform proof states until a complete proof term has been constructed.

Throughout this thesis, proof scripts are written in the same tactic-based style as in the *Software Foundations* series [18]. This style favors short, explicit tactic steps and closely follows the evolution of the proof state, which makes proofs easier to read, especially for readers who are new to Rocq. The *Software Foundations* books also provide a useful introduction to many of the basic tactics used in such scripts. Using this style therefore helps keep the present formalization both readable and accessible.

A small example helps illustrate how Rocq proofs are developed interactively. Consider the statement that conjunction is commutative. In Rocq, this can be written as follows.

Listing 1 A small Rocq proof showing logical conjunction is commutative

```
Theorem conj_comm : forall (A B : Prop), A /\ B -> B /\ A.
Proof.
  intros A B HAB.
  destruct HAB as [HA HB].
  split.
  - exact HB.
  - exact HA.
Qed.
```

The theorem statement is a Gallina term, while the commands `Proof` and `Qed` belong to Rocq's vernacular language. The proof itself is written using tactics. During interactive proof development, Rocq maintains a proof state consisting of the current context of assumptions together with the remaining goal. For the proof above, the proof state evolves as follows.

Listing 2 Intermediate proof states for Listing 1

```
1 goal
============================
forall A B : Prop, A /\ B -> B /\ A
```

After applying `intros A B HAB`, the universally quantified variables `A` and `B`, as well as the hypothesis `HAB : A /\ B`, are moved into the local context. The remaining goal is then to prove `B /\ A` under these assumptions.

```
A, B : Prop
HAB : A /\ B
============================
B /\ A
```

Next, the tactic `destruct HAB as [HA HB]` decomposes the conjunction `A /\ B`. This replaces the single hypothesis `HAB` by two hypotheses, namely `HA : A` and `HB : B`.

```
A, B : Prop
HA : A
HB : B
============================
B /\ A
```

Finally, the tactic `split` uses the fact that the goal is itself a conjunction. It reduces the original goal `B /\ A` to two separate subgoals, one for each conjunct.

```
A, B : Prop
HA : A
HB : B
============================
B
```

```
A, B : Prop
HA : A
HB : B
============================
A
```

These two remaining subgoals are then discharged by `exact HB` and `exact HA`, respectively.

This example makes the different layers of Rocq more concrete. The theorem statement itself is written in Gallina, the proof is guided by tactics such as `intros`, `destruct`, and `split`, and the intermediate proof states record what remains to be shown at each stage. Once the proof is completed, Rocq translates the tactic script into a proof term and checks this term against the theorem statement using the kernel.

This distinction is important for understanding Rocq's guarantees. Although users may rely on Gallina definitions, vernacular commands, tactics, and plugins when building a formalization, the final object checked by Rocq is always a proof term in the core theory. In other words, once a proof has been completed, Rocq reduces correctness to kernel checking of a term against its type [3]. This greatly limits the trusted base of the system. Tactics such as those written in Ltac may help construct proofs, but they do not enlarge the trusted kernel, because any theorem they produce must still be validated by the kernel before it is accepted. As emphasized in the broader literature on proof assistants, this reflects the de Bruijn criterion that the trusted core of a prover should remain as small as possible [3, 12].

This architecture is particularly useful in a verification project such as this. It allows the SAT solver framework to be specified directly in Gallina, while proofs of its properties can be developed interactively with tactics and automation. The resulting correctness guarantees are therefore machine-checked guarantees about the formal objects defined in the development, rather than only about an informal mathematical argument.

2.1 The Equations Plugin

In Rocq, recursive functions are easiest to define when they follow *structural recursion*, that is, when each recursive call is made on a direct subterm of the original input, so that one of its structural components becomes smaller at each step. This is the style directly supported by ordinary `Fixpoint` definitions and works well for many simple functions on inductive datatypes. In this thesis, however, several recursive definitions are more naturally described using a well-founded relation rather than an immediately smaller subterm. For this reason, the Equations plugin [20] is particularly useful. It provides convenient support for such definitions, while still compiling them into ordinary core terms and automatically generating useful equations and proof principles.

A simple example, adopted from [1], is a subtraction-based version of Euclid's algorithm for the greatest common divisor. If one tries to define such a function directly, the recursive calls are performed on arguments such as `(x, y - x)` or `(x - y, y)`. These are intuitively smaller than the original pair, but they are not direct subterms of it, so ordinary structural recursion does not apply. Equations makes it possible to define the function directly at its intended type, while proving termination separately using a well-founded measure. In this case, the natural measure is `x + y`, which strictly decreases in each recursive call.

Listing 3 A well-founded recursive definition with Equations

```
Equations gcd (x y : nat) : nat by wf (x + y) lt :=
gcd 0 y := y;
gcd x 0 := x;
gcd x y with compare x y :=
  | Eq => x;
  | Lt => gcd x (y - x);
  | Gt => gcd (x - y) y.
Next Obligation. lia. Qed.
Next Obligation. lia. Qed.
```

This example illustrates an important advantage of Equations. The function itself can be written in the form that one would naturally use on paper, while the termination argument is handled separately through obligations generated by the `by wf` annotation. In the example above, these obligations are simple arithmetic proof goals, and `lia` is a standard Rocq tactic for solving such linear arithmetic goals over natural numbers and integers. Some rewriting may occasionally be needed, but the resulting definition remains close to the intended algorithm, rather than being distorted to satisfy the syntactic guard condition.

Equations is also useful when proving properties of such functions. Besides the defining equations, it generates an elimination principle adapted to the recursive structure, which can be used through the `funelim` tactic. For example, the following proof shows `gcd x x = x`.

Listing 4 A proof using the elimination principle generated by Equations

```
Lemma gcd_same x : gcd x x = x.
Proof.
  funelim (gcd x x).
  - reflexivity.
  - reflexivity.
  - rewrite compare_lt_iff in Heq. lia.
  - rewrite compare_gt_iff in Heq. lia.
Qed.
```

It is of course possible to define such functions directly using Rocq's well-founded recursion machinery, for example through `Fix`. In practice, however, this often means that proof terms witnessing termination become part of the definition and have to be handled more explicitly. Equations provides a more convenient interface for this situation. It supports dependent pattern matching and structural, mutual, nested, and well-founded recursion, while compiling the result into ordinary core terms and automatically generating useful proof principles, including the defining equations of the function and an associated elimination principle [20].

This convenience is especially helpful here, where recursive definitions and proofs about them are closely connected. At the same time, using Equations also comes with small drawbacks. In particular, definitions introduced through Equations do not always interact as smoothly with simplification tactics such as `simpl` as plain `Fixpoint` definitions do. In practice, proofs often proceed instead by rewriting manually with the equations generated by Equations. Even so, the gain in readability and the better support for well-founded recursion make Equations a natural choice for this formalization.

2.2 The Rocq Formalization

Throughout the thesis, references to formal objects are presented using hyperlink macros, for example . These links are intended as a navigation aid for the reader. They make it possible to move directly from an informal statement in the text to the corresponding formal definition, lemma, or theorem in either the generated documentation of the present formalization or Rocq's library documentation. This style of linking the informal text directly to the formal artifact is inspired by earlier work by Arnoud van der Leer et al. [21]. In this way, the thesis is linked directly to the formal artifact on which its claims are based.

To support reproducibility, the full Rocq formalization is publicly available in the RocqSAT repository at `https://github.com/Vlamonster/RocqSAT`, and generated documentation, including a table of contents for the formalization, is available on GitHub Pages at `https://vlamonster.github.io/RocqSAT/toc`. The development can be built by first installing Opam, tested with version `2.5.0`, then entering the `/code` directory and installing the dependencies with `opam install . -deps-only`, and finally building the project with `opam exec - dune build`. During the development of the formalization, VSRocq was used for interactive proof development, with `vsrocq-language-server` version `2.3.3`.

The formalization depends on `rocq-stdlib 9.0.0`, `rocq-prover 9.0.0`, `rocq-core 9.0.1`, `rocq-runtime 9.0.1`, `rocq-equations 1.3.1+9.0`, and `dune 3.20.2`. In its current form, the development contains 187 lemmas, 7 theorems, 75 definitions, and 14 examples. Of these 75 definitions, 30 are standard Rocq definitions, 36 are Equations definitions, and 9 are inductive Rocq definitions. Altogether, the formalization consists of 3555 lines of code, of which 2403 lines are proof scripts between `Proof.` and `Qed.`

3 Propositional Logic

We begin by recalling the standard definition of satisfiability for propositional formulas. The notions introduced in this section are classical and can be found in standard textbooks on logic and formal reasoning, such as Huth and Ryan's *Logic in Computer Science* [9]. We first introduce the basic syntactic elements used to construct propositional formulas.

3.1 Syntax of Propositional Logic

For the following definitions we will fix a finite set of propositional symbols P.

▶ **Definition 1** (🔥). *Elements $p \in P$ are called **atoms**.*

▶ Remark 2. Since P is finite, there exists a bijection between P and a finite subset of the natural numbers. Consequently, we may identify atoms with elements of $\mathbb{N}$.

A **literal** corresponds to either a positive or a negative occurrence of an atom, enabling formulas to represent both assertions and their negations.

▶ **Definition 3** (🔥, 🔥). *A **literal** over P is either an atom $p \in P$ or its negation $\neg p$, where negation is the involutive function that flips the sign.*

Literals can then be combined into **clauses**.

▶ **Definition 4** (🔥). *A **clause** is a finite disjunction of literals, written $l_1 \vee \ldots \vee l_n$.*

Formulas are often expressed in **conjunctive normal form**. This form is convenient for reasoning about satisfiability and is the standard input for many SAT-solvers. Although this may appear to be a restriction, it does not lose generality, since every propositional formula can be transformed into an equivalent formula in conjunctive normal form.

▶ **Definition 5** (🔥). *A propositional **formula** in **conjunctive normal form** (CNF) is a finite conjunction of clauses, written $c_1 \wedge \ldots \wedge c_k$.*

One such way of transforming formulas into conjunctive normal form, following the methods in [19, 11], is to exhaustively apply the following rules in sequence:

Eliminate $\leftrightarrow$:	$\dfrac{x \leftrightarrow y}{(x \to y) \wedge (y \to x)}$,	Eliminate $\to$:	$\dfrac{x \to y}{\neg x \vee y}$
Push $\neg$ into $\wedge$:	$\dfrac{\neg(x \wedge y)}{\neg x \vee \neg y}$,	Push $\neg$ into $\vee$:	$\dfrac{\neg(x \vee y)}{\neg x \wedge \neg y}$
Eliminate $\neg\neg$:	$\dfrac{\neg\neg x}{x}$,	Distribute $\vee$ over $\wedge$:	$\dfrac{x \vee (y \wedge z)}{(x \vee y) \wedge (x \vee z)}$

We first eliminate all double implications and implications. Next, we push negations inward using De Morgan's laws and eliminate double negations. After these steps, the formula is in what is known as **negation normal form** (NNF). We then repeatedly distribute disjunction over conjunction until the result is a conjunction of clauses, that is, a formula in conjunctive normal form. Because every propositional formula can be transformed in this way, we will use the term **formula** to refer to CNF formulas from now on.

▶ **Example 6.** Consider the formula $\neg(x \leftrightarrow y)$. We transform this into CNF as follows:

$$\begin{aligned}
\neg(x \leftrightarrow y) &= \neg((x \to y) \land (y \to x)) && \text{(eliminate } \leftrightarrow\text{)} \\
&= \neg((\neg x \lor y) \land (\neg y \lor x)) && \text{(eliminate } \to\text{)} \\
&= \neg(\neg x \lor y) \lor \neg(\neg y \lor x) && \text{(push } \neg \text{ into } \land\text{)} \\
&= (\neg\neg x \land \neg y) \lor (\neg\neg y \land \neg x) && \text{(push } \neg \text{ into } \lor\text{)} \\
&= (x \land \neg y) \lor (y \land \neg x) && \text{(eliminate } \neg\neg\text{)} \\
&= ((x \land \neg y) \lor y) \land ((x \land \neg y) \lor \neg x) && \text{(distribute } \lor \text{ over } \land \text{ on left)} \\
&= (x \lor y) \land (\neg y \lor y) \land (x \lor \neg x) \land (\neg y \lor \neg x) && \text{(distribute } \lor \text{ over } \land \text{ on right)} \\
&= (x \lor y) \land (\neg y \lor \neg x) && \text{(simplification).}
\end{aligned}$$

▶ Remark 7. The Rocq formalization represents clauses and formulas as lists, without specifying an evaluation procedure. The evaluation semantics are provided by the definition of **partial assignments** instead.

We now illustrate how a SAT encoding can be used on a concrete problem by showing how it can establish the equivalence of two digital circuits.

▶ **Example 8.** We wish to formally show using SAT that an XOR gate can be implemented using four NAND gates. Figure 1 visualizes this using a composite circuit consisting of the XOR gate, the NAND-only implementation, and a final circuit that checks whether the two outputs differ. To encode this as SAT, we use the fact that a NAND gate with inputs a, b and output c can be written as $(\neg a \lor \neg b) \leftrightarrow c$, while an XOR gate with inputs a, b and output c can be written as $((a \lor b) \land (\neg a \lor \neg b)) \leftrightarrow c$. Using the variables p, q, r, u, v, and o for the intermediate outputs shown in the figure, we obtain the following constraints:

$$\begin{aligned}
\text{XOR circuit: } & ((x \lor y) \land (\neg x \lor \neg y)) \leftrightarrow u \\
\text{NAND circuit: } & (\neg x \lor \neg y) \leftrightarrow p \ \land \ (\neg x \lor \neg p) \leftrightarrow q \ \land \ (\neg y \lor \neg p) \leftrightarrow r \ \land \ (\neg q \lor \neg r) \leftrightarrow v \\
\text{Difference circuit: } & ((u \lor v) \land (\neg u \lor \neg v)) \leftrightarrow o
\end{aligned}$$

Note that o is obtained by applying XOR to the outputs of the two circuits, and is true exactly when they differ. We now transform each of these constraints into CNF using the method from Example 6, and take the conjunction of all resulting CNF formulas together with the unit clause o. The formula is unsatisfiable if and only if there is no input assignment on which the two circuit outputs differ, that is, if and only if the two circuits are equivalent.

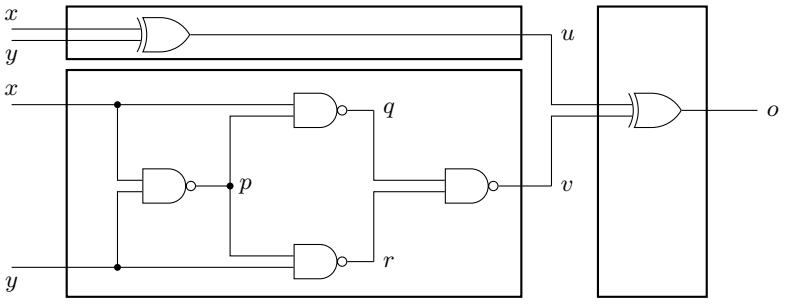

Figure 1 A circuit comparing a direct XOR gate with its NAND-only implementation.

3.2 Evaluation

We first recall the standard notion of partial assignments from the literature.

▶ **Definition 9.** *Let f be a formula. A **partial assignment** m for f is a subset of literals appearing in f such that $\{p, \neg p\} \not\subseteq m$ for all propositional variables $p \in P$.*

While this set-based definition of partial assignments is natural, it is cumbersome for formal verification. We instead represent partial assignments as lists. This choice is motivated by the fact that the `list` type supports well-behaved induction principles, which facilitate formal reasoning, and also allows a partial assignment to be interpreted as a **trace** of assignments.

Each literal is also annotated with a type `Ann`. We add the superscript d to a literal, writing l^d, to indicate that it is a **decision literal**. These annotations will be ignored during evaluation, but are essential for defining and reasoning about the DPLL transition rules.

▶ **Definition 10.** *A **partial assignment** is a list of annotated literals.*

To recover set-like behavior, we require an additional property.

▶ **Definition 11.** *A partial assignment m is **well-formed** in a formula f if it is both **duplicate-free**, so that $\{p, \neg p\} \not\subseteq m$ for all $p \in P$, and **bounded** by f, so that every literal in m appears in f.*

Let m be a partial assignment. We define a three-valued evaluation function

$$[\![\cdot]\!]_m : \texttt{Lit} + \texttt{Clause} + \texttt{CNF} \to \{\texttt{T}, \texttt{F}, \texttt{U}\}.$$

In a slight abuse of notation, we write $l \in m$ to mean that l appears in m *before* any occurrence of $\neg l$, evaluating the list from head to tail. Note that this notation coincides with its standard definition for well-formed partial assignments.

▶ **Definition 12.** *For a literal l, we define*

$$[\![l]\!]_m = \begin{cases} T & \text{if } l \in m, \\ F & \text{if } \neg l \in m, \\ U & \text{otherwise.} \end{cases}$$

▶ **Definition 13.** *For a clause c, we define*

$$[\![c]\!]_m = \begin{cases} T & \text{if some } l \in c \text{ satisfies } [\![l]\!]_m = T, \\ F & \text{if all } l \in c \text{ satisfy } [\![l]\!]_m = F, \\ U & \text{otherwise.} \end{cases}$$

▶ **Definition 14.** *For a CNF formula f, we define*

$$[\![f]\!]_m = \begin{cases} T & \text{if all } c \in f \text{ satisfy } [\![c]\!]_m = T, \\ F & \text{if some } c \in f \text{ satisfies } [\![c]\!]_m = F, \\ U & \text{otherwise.} \end{cases}$$

One benefit of the list-based representation is that evaluation is defined for arbitrary lists of literals, not only for well-formed partial assignments. This lets us reason about more general lists throughout the formalization, while only introducing well-formedness assumptions in the lemmas that require them. By contrast, in a set-based approach, a set containing both x and $\neg x$ cannot be evaluated without imposing an additional well-formedness condition.

▶ **Example 15.** Consider the partial assignment $m = \neg xx$. This partial assignment is not well-formed, since it is not duplicate-free. However, it still provides a well-defined evaluation for all literals, with $[\![x]\!]_m = \mathtt{T}$ and $[\![\neg x]\!]_m = \mathtt{F}$.

▶ **Definition 16.** *A formula f is **satisfiable** if there exists a partial assignment m that satisfies f. Such an assignment is called a **model** of f, written $m \models f$. If no such assignment exists, f is **unsatisfiable**.*

4 Transition Systems

We now turn to the transition-system framework used to describe the DPLL procedure. Rather than presenting DPLL directly as an algorithm, Nieuwenhuis et al. [16] formulate it as a transition system on abstract solver states. This perspective is particularly well suited to formal reasoning, since it separates the logical structure of the procedure from implementation details. The framework introduced below closely follows Section 2.2 of Nieuwenhuis et al., with minor adaptations to notation.

▶ **Definition 17** (). *A **state** is either the distinguished state* `fail` *or a pair of the form* $m \parallel f$*, where* m *is a partial assignment and* f *is a formula.*

Having defined states, we now formalize how one state can evolve into another.

▶ **Definition 18.** *A **transition rule*** $\Longrightarrow$ *is a relation on states. Given two states* s *and* s'*, we say* $s \Longrightarrow s'$ *is a **transition** from* s *to* s'*.*

To model the DPLL procedure, we consider a collection of transition rules. In the following definitions, let T denote such a set of transition rules.

▶ **Definition 19.** *A **transition system*** $\Longrightarrow_T$ *on* T *is a relation on states, where two states are related if one of the transition rules in* T *applies.*

To state later lemmas and theorems, it is useful to consider sequences of transitions, motivating the reflexive–transitive and transitive closures of the transition system.

▶ **Definition 20** (). *Let* $\Longrightarrow_T^*$ *be the reflexive-transitive closure of* $\Longrightarrow_T$*. Sequences of transitions under* $\Longrightarrow_T^*$ *are called **derivations**.*

▶ **Definition 21** (). *Let* $\Longrightarrow_T^+$ *be the transitive closure of* $\Longrightarrow_T$*. Sequences of transitions under* $\Longrightarrow_T^+$ *are called **strict derivations**.*

Finally, to reason about termination, we introduce the notion of a final state.

▶ **Definition 22** (,). *A state is **final** in* T *if there exists no state* s' *such that* $s \Longrightarrow_T s'$*.*

4.1 Transition System for DPLL

We now specialize the general transition-system framework to the DPLL procedure. Following Nieuwenhuis et al. [16], we describe DPLL as a set of transition rules on states. We first give an informal overview of this transition-system view, before defining the rules formally.

In this formulation, the solver gradually extends a partial assignment until it either finds a satisfying assignment or reaches a conflict, applying any rule whose conditions are satisfied. The *UnitPropagate* rule applies when a clause has become unit, meaning that all but one of its literals have been assigned false, so the remaining literal must be assigned true to satisfy the clause. The *Pure* rule assigns a pure literal, that is, a literal whose negation does not occur in the formula, since doing so cannot create a conflict. The *Decide* rule makes a branching choice by assigning a previously undefined literal as a decision literal. If a conflict is reached, then *Backtrack* returns to an earlier decision point, undoes later assignments, and flips that decision, with the flipped literal no longer marked as a decision literal. If no earlier decision remains to be undone, *Fail* applies, indicating that the formula is unsatisfiable.

This is formalized by the following transition rules:

$$\text{UnitPropagate:}\quad m \parallel f \wedge (c \vee l) \implies m\,l \parallel f \wedge (c \vee l) \quad \text{if} \begin{cases} c \text{ is false in } m \\ l \text{ is undefined in } m \end{cases}$$

$$\text{Decide:}\quad m \parallel f \implies m\,l^d \parallel f \quad \text{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } f \\ l \text{ is undefined in } m \end{cases}$$

$$\text{Fail:}\quad m \parallel f \wedge c \implies \texttt{fail} \quad \text{if} \begin{cases} c \text{ is false in } m \\ m \text{ contains no decision literals} \end{cases}$$

$$\text{Backtrack:}\quad m\,l^d\,n \parallel f \wedge c \implies m\,\neg l \parallel f \wedge c \quad \text{if} \begin{cases} c \text{ is false in } m\,l^d\,n \\ n \text{ contains no decision literals} \end{cases}$$

$$\text{Pure:}\quad m \parallel f \implies m\,l \parallel f \quad \text{if} \begin{cases} l \text{ occurs in a clause of } f \\ \neg l \text{ occurs in no clause of } f \\ l \text{ is undefined in } m \end{cases}$$

We now distinguish between two transition systems. We refer to the transition system containing all of the rules above as the **classical DPLL** system. We also define the **base DPLL** system as the subsystem containing only *Decide*, *Fail*, and *Backtrack*. We will later show that this smaller system is still sufficient for defining a complete solving procedure.

▶ **Definition 23** (🔫). *The **classical DPLL** system is a transition system on*

$$C := \{\mathit{UnitPropagate}, \mathit{Decide}, \mathit{Fail}, \mathit{Backtrack}, \mathit{Pure}\}.$$

▶ **Definition 24** (🔫). *The **base DPLL** system is a transition system on*

$$B := \{\mathit{Decide}, \mathit{Fail}, \mathit{Backtrack}\}.$$

▶ Remark 25. Since most of our work concerns the classical DPLL system, the subscript C is often omitted. By slight abuse of notation, $\implies$ is then used to refer both to the transition system and to its underlying transition rules.

We now provide some examples of how this transition system can be applied.

▶ **Example 26.** Consider the formula $f = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$. In the following derivation, we hide any clauses that are already true under the current partial assignment and write them as $(\ldots)$. Then the transition rules can be applied as follows:

$$\begin{aligned} & \emptyset \parallel (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) && \\ \implies & x_1 \parallel \ldots \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) && \text{(Pure)} \\ \implies & x_1 x_2^d \parallel \ldots \wedge (\neg x_2 \vee \neg x_3) && \text{(Decide)} \\ \implies & x_1 x_2^d \neg x_3 \parallel \ldots && \text{(UnitPropagate)} \end{aligned}$$

The first step applies *Pure*, since x_1 occurs only positively in the formula. The second step applies *Decide*, choosing x_2 as a decision literal. After this, the clause $(\neg x_2 \vee \neg x_3)$ becomes unit, so *UnitPropagate* assigns $\neg x_3$. The resulting state is final, since every remaining clause is already satisfied and no rule applies. Since this final state is not `fail`, Theorem 28 implies that the partial assignment $x_1 x_2^d \neg x_3$ is a model of f. This can be easily verified.

▶ **Example 27.** Consider the formula $f = x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$. Again, we replace clauses already true under the current partial assignment by $(\ldots)$. Then the transition rules can be applied as follows:

$$\begin{aligned}
& \emptyset \parallel x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) && \\
\Longrightarrow\ & x_1^d \parallel \ldots \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) && \text{(Decide)} \\
\Longrightarrow\ & x_1^d x_2 \parallel \ldots \wedge (\neg x_1 \vee \neg x_2) && \text{(UnitPropagate)} \\
\Longrightarrow\ & \neg x_1 \parallel x_1 \wedge \ldots && \text{(Backtrack)} \\
\Longrightarrow\ & \texttt{fail} && \text{(Fail)}
\end{aligned}$$

The first step applies *Decide*, choosing x_1 as a decision literal. After this, the clause $(\neg x_1 \vee x_2)$ becomes unit, so *UnitPropagate* assigns x_2. This makes the clause $(\neg x_1 \vee \neg x_2)$ false, so *Backtrack* undoes the assignments made after the last decision and flips that decision, yielding the state $\neg x_1 \parallel x_1 \wedge \ldots$. Since this state contains no decision literals and the clause x_1 is false, the *Fail* rule applies. Since we derived `fail`, Theorem 45 implies f is unsatisfiable.

4.2 Correctness

We show that the classical DPLL transition system is sound and complete with respect to satisfiability. In particular, we prove that the existence of a derivation from the initial state to a final state precisely characterizes whether a formula is satisfiable or unsatisfiable.

We begin with the observation that every final non-fail state yields a model of the formula. This corresponds to part of Lemma 2.9 and Theorem 2.12 of Nieuwenhuis et al. [16].

▶ **Theorem 28** (`final_model`). *Let m be a partial assignment and f a formula such that the state $m \parallel f$ is well-formed. If $m \parallel f$ is a final state, then m is a model of f.*

Proof. Assume that $m \parallel f$ is a final state. We show that m is a model of f by case analysis on $[\![f]\!]_m$. If $[\![f]\!]_m = \mathtt{T}$, the result is immediate by definition of a model. If $[\![f]\!]_m = \mathtt{F}$, there exists a conflicting clause $c \in f$; if m contains a decision literal, the backtrack rule applies, otherwise the fail rule applies. In either case, a transition is possible, contradicting the finality of $m \parallel f$. Finally, if $[\![f]\!]_m = \mathtt{U}$, there exists a clause $c \in f$ containing an unassigned literal $l \in c$, and the decide rule can be applied to l, again contradicting the finality of $m \parallel f$. Therefore, the only possibility is $[\![f]\!]_m = \mathtt{T}$, and m is a model of f. ◀

Next, we show that if there exists a partial assignment m satisfying f, then there exists a partial assignment m' such that there is a derivation from the initial state $\emptyset \parallel f$ to a final state $m' \parallel f$. To establish this, we use a process called **normalization**. This process has four steps. First, we **totalize** the partial assignment, so that every literal occurring in f is defined. Next, we **convert propagation annotations** to decision annotations. We then **bound** the partial assignment by f, removing any literals whose atoms do not occur in f. Finally, we **deduplicate** the partial assignment, making it duplicate-free.

▶ **Example 29** (normalization). Consider the formula $f = (x_1 \vee x_2) \wedge x_3$ together with the partial assignment $m = \neg x_1 x_1 x_3^d x_4$. We illustrate normalization by applying its steps to m.

$$\begin{aligned}
\neg x_1 x_1 x_3^d x_4 &\longrightarrow \neg x_1 x_1 x_3^d x_4 x_2 && \text{(totalize)} \\
&\longrightarrow \neg x_1^d x_1^d x_3^d x_4^d x_2^d && \text{(convert propagations)} \\
&\longrightarrow \neg x_1^d x_1^d x_3^d x_2^d && \text{(bound)} \\
&\longrightarrow x_1^d x_3^d x_2^d && \text{(dedupe)}
\end{aligned}$$

Since the original partial assignment m is a model of f, this example also illustrates one of the invariants of the normalization procedure, namely that normalization is model preserving.

To show that the normalized state can be reached from the initial state and is final, we first establish the invariants of this normalization procedure. Each step establishes a new invariant that is preserved by all later steps, which gives rise to a number of intermediate lemmas. For brevity, we only state the results for the complete normalization procedure.

First, normalization preserves the property of being a model of f.

▶ **Lemma 30** (`normalize_f`). *Let m be a partial assignment and f be a formula. If m satisfies f, and m' is obtained by normalizing m, then m' also satisfies f.*

After totalization, every literal occurring in f is defined.

▶ **Lemma 31** (`normalize_all_def`). *Let m be a partial assignment and f be a formula. If m' is obtained by normalizing m, then every literal in f is defined in m'.*

After converting all annotations, every literal is a decision literal.

▶ **Lemma 32** (`normalize_only_dec`). *Let m be a partial assignment. If m' is obtained by normalizing m, then every literal in m' is a decision literal.*

After bounding, the partial assignment is bounded by f.

▶ **Lemma 33** (`normalize_bounded`). *Let m be a partial assignment and f be a formula. If m' is obtained by normalizing m, then m' is bounded by f.*

Finally, after deduplication, the partial assignment is duplicate-free.

▶ **Lemma 34** (`normalize_no_duplicates`). *Let m be a partial assignment. If m' is obtained by normalizing m, then m' is duplicate-free.*

It remains to show that a derivation to the normalized state exists. For this, we first prove an auxiliary lemma showing that every well-formed partial assignment consisting only of decision literals can be reached from the initial state $\emptyset \parallel f$.

▶ **Lemma 35** (`normalize_derivation_aux`). *Let m be a partial assignment and f be a formula. If m is well-formed in f, and every literal in m is a decision literal, then there exists a derivation from the initial state $\emptyset \parallel f$ to the state $m \parallel f$.*

Proof. We proceed by induction on the partial assignment m. If m is empty, then the result is immediate, since the initial state is exactly $\emptyset \parallel f$. Now suppose m is obtained by extending a smaller partial assignment with one additional annotated literal. Since every literal in m is a decision literal, this annotation must be a decision annotation. Moreover, well-formedness of m implies that the smaller partial assignment is also well-formed. By the induction hypothesis, there exists a derivation from the initial state $\emptyset \parallel f$ to the state corresponding to this smaller partial assignment. It remains to show that the additional literal can be added by one application of *Decide*. Since m is bounded by f, the literal occurs in a clause of f. Since m is duplicate-free, the literal is undefined in the smaller partial assignment. Therefore the conditions of the *Decide* rule are satisfied, and we can extend the derivation by one more step to obtain a derivation from $\emptyset \parallel f$ to $m \parallel f$. ◀

We obtain the desired result by specializing this lemma to the normalized partial assignment.

▶ **Corollary 36** (`normalize_derivation`). *Let m be a partial assignment and f be a formula. If m' is obtained by normalizing m, then there exists a derivation from the initial state $\emptyset \parallel f$ to the state $m' \parallel f$.*

Proof. By Lemmas 33 and 34, the normalized partial assignment m' is well-formed in f. By Lemma 32, every literal in m' is a decision literal. The result then follows from Lemma 35. ◀

We now show that satisfiability is reflected in the existence of a derivation to a final state. The forward direction follows directly from Theorem 28. The converse direction, however, does not appear in this form in Nieuwenhuis et al. [16], and is proved here using the normalization procedure introduced above.

▶ **Theorem 37** (`final_sat_refl`). *Let f be a formula. There exists a derivation from the initial state $\emptyset \parallel f$ to some final state $m \parallel f$ if and only if f is satisfiable.*

Proof. The forward direction follows from Theorem 28. For the reverse direction, assume that f is satisfiable. Then there exists a partial assignment m satisfying f. Let m' be obtained by normalizing m. By Lemma 30, m' also satisfies f, and by Corollary 36, there exists a derivation from the initial state $\emptyset \parallel f$ to the state $m' \parallel f$. It remains to show that $m' \parallel f$ is final. If *Fail* or *Backtrack* applied, then some clause of f would be false in m', contradicting that m' satisfies f. If *UnitPropagate*, *Decide*, or *Pure* applied, then some literal in f would be undefined in m', contradicting Lemma 31. Thus $m' \parallel f$ is final. ◀

There is an analogous theorem for unsatisfiability, but before stating and proving it, we first introduce the notions of **model-preserving literals** and **locally model-preserving literals**, which we will need in order to define **entailment**.

▶ **Definition 38.** *Let f be a formula and l a literal. We say that l is **model-preserving** in f if, for every partial assignment m' satisfying f, the partial assignment $m'l$ satisfies f. For a partial assignment m, we say that l is **locally model-preserving** in f relative to m if, for every partial assignment m' satisfying both m and f, the partial assignment $m'l$ satisfies f.*

We can now define **entailment**. Intuitively, this property expresses that propagated literals appearing in the tails of partial assignments derived by the transition rules are logical consequences of the literals that precede them.

▶ **Definition 39**. *Let m be a partial assignment and f a formula. We say that f **entails** m, or that m is **entailed** by f, if one of the following holds:*

1. *Every model of f satisfies m, and m contains no decision literals.*
2. *The partial assignment m can be written as $m'l^d n$, where every model satisfying $m'l^d$ satisfies n, the tail n contains no decision literals, and m' is entailed by f.*
3. *The partial assignment m can be written as $m'ln$, where l is model-preserving in f, every model satisfying $m'l$ satisfies n, n contains no decision literals, and m' is entailed by f.*

This definition is loosely based on the one given by Nieuwenhuis et al. [16], with the addition of the third case involving model-preserving literals. This extra case is needed to account for the *Pure* rule in our transition system. One could go a step further and strengthen the classical *Pure* rule by considering a literal pure whenever it appears with only one polarity among the clauses that remain undefined under a given partial assignment. In that setting, the third case would need to use locally model-preserving literals instead.

We now wish to relate entailment to unsatisfiability. For this, we show two things. Firstly, entailment is preserved by the transition system. Secondly, for partial assignments containing no decision literals, entailment coincides with logical entailment. We show the second first.

▶ **Lemma 40** (`entailment`). *Let m and m' be partial assignments and f a formula. If m contains no decisions, m' is a model of f, and f entails m, then $m'm$ is also a model of f.*

Proof. We proceed by induction on the proposition that f entails m. In the first case, the claim follows directly from the definition. In the second case, m is of the form $m''l^d n$. This contradicts the assumption that m contains no decision literals. In the third case, m is of the form $m''ln$. By the induction hypothesis, $m'm''$ is a model of f. Since l is model-preserving, it follows that $m'm''l$ is also a model of f. Moreover, every model satisfying $m''l$ satisfies n, so $m'm''l$ also satisfies n. Therefore $m'm''ln = m'm$ is a model of f. ◀

▶ Remark 41. The conclusion of the `entailment` lemma may seem unusual, since it gives no guarantee that $m'm$ is well-formed. This is not a problem for the results that follow. In fact, reasoning about such extensions will be useful in later proofs.

The next lemma shows that entailment is preserved under the transition system. Since a full formal proof depends on a number of technical lemmas, we only give some intuition for the argument here. The proof proceeds by case analysis on the applied transition rule. For *Fail*, the assumptions lead directly to a contradiction. For *Decide* and *Pure*, the second and third cases of the entailment definition apply directly. The argument for the *Backtrack* and *UnitPropagate* rules is more subtle, since they require unpacking how the original entailment was constructed. The key idea is that repeatedly unpacking this construction must eventually lead either to a contradiction or to a point where one of the entailment cases applies.

▶ **Lemma 42** (`trans_entails`). *Let m and m' be partial assignments and f a formula. If there is a transition from $m \parallel f$ to $m' \parallel f$, and f entails m, then f also entails m'.*

It follows immediately that the same holds not only for transitions, but also for derivations.

▶ **Lemma 43** (`derivation_entails`). *Let m and m' be partial assignments and f a formula. If there is a derivation from $m \parallel f$ to $m' \parallel f$, and f entails m, then f entails m'.*

Before the main result, we first state one final lemma about derivations to `fail`.

▶ **Lemma 44** (`fail_predecessor`). *Let m be a partial assignment and f a formula. If there exists a derivation from $m \parallel f$ to `fail`, then there exists a partial assignment m' such that there is a derivation from $m \parallel f$ to $m' \parallel f$, and a transition from $m' \parallel f$ to `fail`.*

We now have all the required tools needed to show how unsatisfiability is characterized by the `fail` state. The reverse direction relies on a result we will show in the next subsection. This theorem corresponds closely to part (1) of Theorem 2.12 in Nieuwenhuis et al. [16].

▶ **Theorem 45** (`final_unsat_refl`). *Let f be a formula. There exists a derivation from the initial state $\emptyset \parallel f$ to the `fail` state if and only if f is unsatisfiable.*

Proof. For the forward direction, assume there is a derivation from the initial state $\emptyset \parallel f$ to `fail`. By Lemma 44, there exists an intermediate partial assignment m such that there is a derivation from $\emptyset \parallel f$ to $m \parallel f$, followed by a transition from $m \parallel f$ to `fail`. By the *Fail* rule, some clause c of f is false in m, and m contains no decision literals. Since the empty partial assignment is trivially entailed by f, Lemma 43 gives that f entails m. Now suppose that f is satisfiable, and let m' be a model of f. Since m contains no decision literals, the entailment lemma implies that $m'm$ is also a model of f. But the clause c, being false in m, is also false in $m'm$, contradicting that $m'm$ is a model of f. Hence f is unsatisfiable.

For the reverse direction, assume that f is unsatisfiable. By contraposition of Theorem 37, there is no derivation from the initial state $\emptyset \parallel f$ to a final state of the form $m \parallel f$. On the

other hand, we will show in the termination subsection that some final state is reachable from $\emptyset \parallel f$. Since the formula component is preserved along derivations, this final state must be either of the form $m \parallel f$ or `fail`. The first case is impossible, so the reachable final state must be `fail`. Therefore there exists a derivation from the initial state $\emptyset \parallel f$ to `fail`. ◀

▶ Remark 46. If one wishes to extend the transition system with *Learn* and *Forget* rules, the formula component of states may change. In that setting, a stronger notion of **equivalence** would be needed to prove this result.

4.3 Termination

To show termination of any procedure based on the transition system introduced above, we use the standard notions of **accessibility** and **well-foundedness**. Intuitively, an element is accessible if all elements below it are themselves accessible, where the notion of below is determined by the relation under consideration. A relation is well-founded if every element is accessible, that is, if accessibility can always be established by a finite proof. This provides a convenient way to prove termination, since once the transition relation is shown to be well-founded, every sequence of transitions must eventually terminate. In the Rocq standard library, these notions are represented by the propositions `Acc` and `well_founded`.

Listing 5 The `Acc` proposition from the Rocq standard library

```
Inductive Acc (x : A) : Prop :=
| Acc_intro : (forall y : A, R y x -> Acc y) -> Acc x.
```

Listing 6 The `well_founded` proposition from the Rocq standard library

```
Definition well_founded := forall a : A, Acc a.
```

In practice, one rarely proves well-foundedness directly from the definitions above. Instead, one typically starts from standard well-founded relations already available in the Rocq standard library and then uses standard constructions to build more complex well-founded relations from them. In particular, the strict order $<$ on the natural numbers is well-founded, and the standard library provides a proof of this fact.

For the termination arguments below, we use several standard definitions concerning relations from the standard library. We first introduce these definitions, and then state the corresponding lemmas showing that the associated constructions preserve well-foundedness.

▶ **Definition 47** (`inclusion`). *Let A be a type, and let R, S be relations on A. We say that R is **included** in S if, for all $x, y : A$, $x\ R\ y$ implies $x\ S\ y$.*

▶ **Definition 48** (`slexprod`). *Let A and B be types, let R be a relation on A, and let S be a relation on B. The **lexicographic product** of R and S, denoted (R, S), is the relation on $A \times B$ that relates (x, y) to (x', y') if either $x\ R\ x'$, or $x = x'$ and $y\ S\ y'$.*

▶ **Definition 49** (`clos_trans`). *Let A be a type, and let R be a relation on A. The **transitive closure** of R, denoted R^+, is the relation on A defined as follows:*

- *for all $x, y : A$, if $x\ R\ y$, then $x\ R^+\ y$;*
- *for all $x, y, z : A$, if $x\ R^+\ y$ and $y\ R^+\ z$, then $x\ R^+\ z$.*

▶ **Definition 50** (`union`). *Let A be a type, and let R, S be relations on A. The **union** of R and S, denoted $R + S$, is the relation on A that relates x and y if $x\ R\ y$ or $x\ S\ y$.*

▶ **Definition 51** (`commut`). *Let A be a type, and let R, S be relations on A. We say that R and S **commute** if, for all $x, y, z : A$, whenever $y\ R\ x$ and $z\ S\ y$, there exists some y' such that $y'\ S\ x$ and $z\ R\ y'$.*

We will use the following standard lemmas stating that these constructions preserve well-foundedness.

▶ **Lemma 52** (`wf_inverse_image`). *Let A and B be types, let $f : A \to B$ be a function, and let R be a relation on B. Define a relation R_f on A by $x\ R_f\ y$ if and only if $f(x)\ R\ f(y)$. If R is well-founded, then R_f is well-founded.*

▶ **Lemma 53** (`wf_incl`). *If a relation R is included in a relation S, then well-foundedness of S implies well-foundedness of R.*

▶ **Lemma 54** (`wf_slexprod`). *If R and S are well-founded, then their lexicographic product is also well-founded.*

▶ **Lemma 55** (`wf_clos_trans`). *If R is well-founded, then its transitive closure is also well-founded.*

▶ **Lemma 56** (`wf_union`). *If R and S commute, then well-foundedness of R and S implies well-foundedness of their union.*

We now introduce a well-founded relation on bounded lists of natural numbers that will be used in the termination argument.

▶ **Definition 57** (`PrefixLt`). *Let t be a natural number. We define $\mathtt{PrefixLt}_t$ to be the relation on lists of natural numbers of length at most t. We say that m is smaller than m' in the **prefix order** if m and m' agree on a common prefix and, at the first position where they differ, the entry of m is smaller than the corresponding entry of m'.*

Note that this is not quite the standard lexicographic order on lists. In the standard lexicographic order, a list can be smaller simply because it is shorter. By contrast, $\mathtt{PrefixLt}_t$ only compares lists at the first position where they differ.

It follows from the well-foundedness of $<$ on the natural numbers and the lemmas above, in particular Lemma 54, that this relation is well-founded for every choice of t.

▶ **Lemma 58** (`wf_prefix_lt`). *For every natural number t, $\mathtt{PrefixLt}_t$ is well-founded.*

We now formalize the termination argument used by Niewenhuis et al. to show that the classical DPLL system is well-founded. The idea is to equip states with two measures, `score` and `score_total`, and to show that every transition rule decreases with respect to one of them. More precisely, we will prove that all transition rules except *Fail* decrease with respect to the lexicographic product of these measures, together with the prefix order and the standard order $<$ on the natural numbers.

▶ **Definition 59** (`score`). *Let f be a formula with t distinct variables, and let m be a well-formed partial assignment in f of the form*

$$m = m_0\, l_1^d\, m_1\, \ldots\, l_n^d\, m_n,$$

where, for each i, the segment m_i contains no decision literals. The `score` of m is the list

$$t - \mathtt{length}\ m_0\ ::\ t - \mathtt{length}\ m_1\ ::\ \ldots\ ::\ t - \mathtt{length}\ m_n,$$

padded with additional occurrences of t at the end until the resulting list has length t.

▶ **Definition 60** (`score_total`). *Let f be a formula with t distinct variables, and let m be a partial assignment. The `score_total` of m is $t - \mathtt{length}\ m$.*

We briefly illustrate these measures on the transition rules from Example 27. For convenience, we write `TotalLt` for the standard order $<$ on the natural numbers. The key point is that along each transition, the measure `score` decreases with respect to `PrefixLt`, and if `score` remains unchanged, then `score_total` decreases with respect to `TotalLt`.

▶ **Example 61.** Consider the formula $f = x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$. This formula has two distinct variables, so $t = 2$. Following the derived states from Example 27, we compute the corresponding values of `score` and `score_total` to be:

$$\begin{array}{llllr}
\texttt{score}_f\ \emptyset & = 2 :: 2, & \texttt{score_total}_f\ \emptyset & = 2 & \\
\texttt{score}_f\ x_1^d & = 2 :: 2, & \texttt{score_total}_f\ x_1^d & = 1 & \text{(Decide)} \\
\texttt{score}_f\ x_1^d x_2 & = 2 :: 1, & \texttt{score_total}_f\ x_1^d x_2 & = 0 & \text{(UnitPropagate)} \\
\texttt{score}_f\ \neg x_1 & = 1 :: 2, & \texttt{score_total}_f\ \neg x_1 & = 1 & \text{(Backtrack)}
\end{array}$$

The *Decide* step leaves `score` unchanged and decreases `score_total`. By contrast, the *UnitPropagate* and *Backtrack* steps decrease `score` itself.

To complete the termination argument, it remains to account for the state `fail`. For this purpose, we introduce the relation `FailLt`, which holds exactly when the left-hand side is `fail` and the right-hand side is a non-fail state. In this way, `fail` becomes smaller than every non-fail state, which leads to the following relation on states.

▶ **Definition 62** (`StateLt`). *Let f be a formula. We define the relation $\ll_f$ on states by*

$$\ll_f \;=\; \mathit{FailLt} + (\mathit{PrefixLt}_{\mathit{score}},\ \mathit{TotalLt}_{\mathit{score\_total}}),$$

where the bound for $\mathit{PrefixLt}_{\mathit{score}}$ is given by the number of distinct atoms in f.

It follows directly from the lemmas above, in particular Lemmas 56, 54, and 52, together with the well-foundedness of `FailLt`, `PrefixLt`, and `TotalLt`, that this relation is well-founded.

▶ **Lemma 63** (`wf_state_lt`). *For every formula f, the relation $\ll_f$ is well-founded.*

Finally, we show that $\Longrightarrow$ is well-founded. The key observation is that, for every state, there exists a formula f such that the transition relation is included in $\ll_f$. The claim then follows from Lemma 53 together with the well-foundedness of $\ll_f$ established above.

▶ **Theorem 64** (`wf_trans`). *The relation $\Longrightarrow$ is well-founded.*

5 Solve Procedures

The transition system from the previous section provides an abstract description of DPLL and establishes its key properties, in particular correctness, completeness, and termination. We now show how to turn this abstract system into a solving procedure. The main idea is to equip the transition system with a *strategy* that determines, for each non-final state, which transition to take next. We then obtain a solver based on such a strategy by repeatedly applying it until a final state is reached.

5.1 Abstract Solve Procedure

We formalize this by introducing strategies as functions from states to optional successor states, where `None` indicates a final state, and that satisfy the following conditions.

▶ **Definition 65** (). *A function* $\mathit{next} : \mathit{State} \to \mathit{option}\ \mathit{State}$ *is a* ***strategy*** *if it satisfies:*

- $\mathit{next}\ \mathit{fail} = \mathit{None}$.
- *For all states* s, *if* $\mathit{next}\ s = \mathit{None}$, *then* s *is final in* B.
- *For all states* s *and* s', *if* $\mathit{next}\ s = \mathit{Some}\ s'$, *then* $s \Longrightarrow^{+} s'$.

We highlight two aspects of this definition that will be important in what follows.

First, we require only that a strategy step follows a strict derivation, rather than a single transition. This allows a strategy to combine several transition steps into one, for example by making multiple decisions at once. Although efficiency is not the main focus of this thesis, this flexibility is still useful, since it allows the abstract notion of strategy to cover a broader range of solving procedures.

Second, the requirement that `next` returns `None` only on states that are final in B is sufficient because the rules in B already capture the essential search structure of DPLL. The additional rules *UnitPropagate* and *Pure* do not lead to fundamentally new search states. Rather, they only allow certain literals to be assigned by propagation instead of by explicit decision, which can speed up the search. Thus, being final in B is equivalent to being final in C, as stated in the following lemma.

▶ **Lemma 66** (`final__final_b`). *A state is final in* C *if and only if it is final in* B.

Proof. For the forward direction, every rule of B is also a rule of C. Therefore, any transition available in B is also available in C, so finality in C implies finality in B.

For the converse direction, assume a state is final in B, and argue by contradiction that it must also be final in C. If it were not final in C, then some rule of C would apply. If this rule were *Fail*, *Decide*, or *Backtrack*, then it would already give a transition in B, contradicting finality in B. If it were *UnitPropagate* or *Pure*, then the corresponding literal is undefined and occurs in the formula, so the *Decide* rule can be applied to that same literal, again yielding a transition in B. This again contradicts finality in B. ◀

Given a strategy `next`, we define the induced solver by iterating `next` from the initial state until a final state is reached. This solver is well-defined, since its iteration must terminate. Each application of `next` yields a step in the transitive closure $\Longrightarrow^{+}$ of the transition relation. Because $\Longrightarrow$ is well-founded by Theorem 64, Lemma 55 shows that $\Longrightarrow^{+}$ is well-founded as well. Hence the iteration of `next` cannot produce an infinite sequence of states.

▶ **Definition 67** (`solve`). *Let* next *be a strategy. We define* $\mathit{solve}(\mathit{next}) : \mathit{CNF} \to \mathit{State}$ *to be the solver that maps a formula* f *to the state obtained by repeatedly applying* next *to the initial state* $\emptyset \parallel f$ *until* next *returns* None.

By Lemma 66, any state returned by `solve` is final. It therefore follows from Theorems 28 and 45 that the solver behaves as expected. If `solve` returns `fail`, then the formula f is unsatisfiable, while if it returns a non-`fail` state of the form $m \parallel f$, then m is a model of f.

5.2 Concrete Solve Procedure

We now instantiate the abstract notion of strategy by defining a concrete function `next_state`. This function covers all transition rules of the classical DPLL system except for *Pure*. It first checks whether the current state contains a conflict. If so, it determines whether there is a decision literal to backtrack to. If there is none, it applies *Fail*. Otherwise, it applies *Backtrack*. If no conflict is present, it tries to find a unit clause, and if one is found, it applies *UnitPropagate*. Only if no conflict and no unit clause are found does it apply *Decide*. In this way, `next_state` yields a deterministic instance of the abstract strategy introduced above.

The following listing gives a simplified overview of this function, with the proof objects witnessing well-formedness omitted for readability. Here, `++p` denotes concatenation with a propagated literal, and `++d` denotes concatenation with a decision literal.

Listing 7 The concrete strategy `next_state`

```
Equations next_state (s: State): option State :=
next_state fail        := None;
next_state (state m f) :=
  match find_conflict m f with
  | Some c =>
    match split_last_decision m with
    | None         => Some fail
    | Some (m', l) => Some (state (m' ++p ¬l) f)
    end
  | _ =>
    match find_unit m f with
    | Some (c, l) => Some (state (m ++p l) f)
    | None        =>
      match find_decision m f with
      | Some l => Some (state (m ++d l) f)
      | None   => None
      end
    end
  end.
```

It remains to show that `next_state` is indeed a strategy. For this, we verify two properties. First, `next_state` is sound with respect to $\Longrightarrow$, meaning that whenever it returns a successor state, that successor is obtained by a single transition step, and hence by a strict derivation. Second, `next_state` is complete with respect to $\Longrightarrow_B$, meaning that whenever a transition in the base DPLL system is possible, `next_state` returns some successor state. Together with the fact that `next_state fail` = `None`, this is enough to conclude that `next_state` satisfies the definition of a strategy. These two properties are stated in the following lemmas.

▶ **Lemma 68** (`next_state_sound`). *Let s and s' be states. If `next_state` s = `Some` s', then there is a transition from s to s'.*

▶ **Lemma 69** (`next_state_exists`). *Let s and s' be states. If there is a transition from s to s' in the base DPLL system, then there exists a state s'' such that `next_state` s = `Some` s''.*

5.3 Extraction to OCaml

In addition to the formalization of the transition system and solving procedures, the concrete solver was also extracted to OCaml using Rocq's extraction mechanism. For this purpose, the basic datatypes `bool`, `list`, and `nat` were mapped to their OCaml counterparts using the extraction directives shown in Listing 8.

Listing 8 Extraction directives used for the OCaml solver

```
Extract Inductive bool => "bool" [ "true" "false" ].

Extract Inductive list => "list" [ "[]" "(::)" ].

Extract Inductive nat => "int"
  [ "0" "(fun x -> x + 1)" ]
  "(fun zero succ n -> if n=0 then zero () else succ (n-1))".
```

The extracted solver, together with a small runner and instructions for how to execute it, is available in the repository. To gain some confidence in the extracted code, it was tested on a number of simple SAT instances, many of them taken from Gregory Duck's collection of SAT examples [6]. For all instances on which the solver finished, it produced the correct result. The largest example it was able to handle was `zebra.cnf`, which has 155 variables and 1135 clauses. Of course, the size of a SAT instance does not necessarily correspond directly to its difficulty, so this number should only be taken as a rough indication.

The current extracted solver does not scale well to larger instances. The main expected reason is the choice of data structures in the formalization and in the concrete strategy. Currently, many operations require traversing lists, which leads to linear-time access. A more efficient implementation would use data structures supporting constant-time lookup, such as arrays or similarly indexed data structures. This would likely improve the practical performance substantially, but was beyond the scope of this thesis.

6 Discussion

The results of this thesis should be understood as an abstract verified core for SAT solving. The formalization establishes correctness, completeness, and termination for the DPLL transition system, and derives both an abstract solver and a concrete strategy from it. This gives strong guarantees about the logical correctness of the procedure, but it does not yet amount to a verification of the behavior of modern industrial SAT solvers. In particular, this thesis focuses on an abstract transition-system presentation and on a simple verified strategy, rather than on the highly optimized algorithms and data structures used in practice.

A first limitation is therefore the scope of the transition system itself. The formalization covers classical DPLL together with the pure literal rule, but it does not yet include extensions that are central to modern SAT solving, such as non-chronological backtracking, clause learning, clause forgetting, and restarts. These mechanisms are precisely what make modern CDCL-based solvers effective on large benchmark instances. Relatedly, this thesis does not yet cover the second part of Nieuwenhuis et al. [16], namely the extension from SAT to DPLL(T). The current formalization can therefore be seen as addressing the propositional core of their work, while leaving the SMT-oriented generalization for future work.

A second limitation concerns the concrete strategy derived in Chapter 5. Although it is formally verified as a strategy, it is intentionally simple and is not designed for performance. In particular, the function `next_state` does not implement the *Pure* rule. This is partly a consequence of the way the project evolved, since the scope of the thesis was expanded later to include formalization of the pure literal rule at the abstract level. The resulting gap does not affect the correctness of the solver, but it does show that the concrete strategy currently lags behind the abstract transition system on which it is based.

More generally, no serious experimental evaluation of the extracted solver has been carried out. Apart from testing correctness on small SAT instances, the thesis does not investigate practical performance, memory usage, or scalability. This is consistent with the main goal of the work, which is formal verification rather than implementation efficiency, but it also means that the current solver should be understood as a verified proof of concept rather than as a practically competitive SAT solver.

There are also limitations in the structure of the formalization itself. Currently, transition systems are treated as large relations rather than as modular collections of transition rules. This is sufficient for the results proved here, but it tends to produce large proofs that proceed by repeated case analysis over all rules at once. A more modular organization, for example in the style used by Marić [13], could make the formalization easier to extend and maintain by splitting larger arguments into smaller lemmas for individual rules.

Finally, the notion of solving procedure used in this thesis could be generalized further. At present, strategies operate only on states. A more abstract design would let them operate on a product of a state and some additional context carrying auxiliary information about the current solve. Such a context could be used to model preprocessing steps or to support more efficient bookkeeping. For example, one could check for pure literals only at the start of a solve and then record in the context that this preprocessing step has already been performed. More generally, such an approach could support strategies that rely on auxiliary data structures for evaluating literals, clauses, or formulas more efficiently, while still fitting into the abstract framework developed here.

7 Conclusion and Future Work

In this thesis, we formalized the DPLL transition system of Nieuwenhuis et al. [16] in Rocq. The formalization covers the core correctness, completeness, and termination results for DPLL, and extends the original development by also formalizing the pure literal rule. Together, these results show that the transition-system presentation of DPLL can be mechanized in an interactive theorem prover while preserving the intended semantic guarantees of the abstract solver.

There are several natural directions for extending this work. One possibility is to enrich the transition system with additional rules that bring it closer to modern SAT solvers [13, 4]. In particular, the backjump rule, also known as non-chronological backtracking, could be formalized. This would naturally be accompanied by a learn rule, which adds clauses derived during conflict analysis, and by a forget rule for removing learned clauses. As in Blanchette et al. [4], learn and backjump are best treated together, since combining them ensures that the solver continues to make progress after new clauses are learned.

Another useful extension would be to add a restart rule, allowing the solver to abandon the current search branch and resume from the root. To preserve termination, such a rule would need to be restricted by an unbounded restart policy. As observed by Blanchette et al. [4], this already suffices, in contrast to the stronger requirement in Nieuwenhuis et al. [16] that the minimum number of transition steps between successive restarts be strictly increasing.

Beyond such extensions of the SAT transition system itself, this formalization could also serve as a basis for a verified SMT solver. Since the framework of Nieuwenhuis et al. [16] is designed to generalize from SAT to DPLL(T), extending the current Rocq formalization in this direction would be a natural next step.

Finally, future work could improve the concrete solving strategies derived from the transition system. The current strategy is sufficient for correctness, but it is still far from the techniques used in practical SAT solvers. Possible improvements include implementing the two-watched-literal scheme for efficient unit propagation and the VSIDS branching heuristic for choosing decision literals [15]. Such extensions would make the verified solver more representative of modern SAT-solving practice while retaining the formal guarantees established in this thesis.

References

1 Equations documentation. https://rocq-prover.org/docs/equations-docs, 2025. Accessed: 2025-11-28.

2 Rocq: The rocq prover. https://github.com/rocq-prover/rocq, 2025. Accessed: 2025-12-08.

3 The rocq prover reference manual, version 9.2.0. https://rocq-prover.org/doc/V9.2.0/refman, 2026. Accessed: 2026-05-07.

4 Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified sat solver framework with learn, forget, restart, and incrementality: Jc blanchette et al. *Journal of automated reasoning*, 61(1):333–365, 2018.

5 Ashish Darbari, Bernd Fischer, and Joao Marques-Silva. Industrial-strength certified sat solving through verified sat proof checking. In *International Colloquium on Theoretical Aspects of Computing*, pages 260–274. Springer, 2010.

6 Gregory Duck. Javascript sat solver. https://www.comp.nus.edu.sg/~gregory/sat/. Accessed: 2025-12-22.

7 Marijn Heule. Schur number five. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

8 Marijn JH Heule, Oliver Kullmann, and Victor W Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 228–245. Springer, 2016.

9 Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2 edition, 2004.

10 Richard M Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, pages 219–241. Springer, 2009.

11 Mark Verus Lawson. *A first course in logic*. CRC Press, 2018.

12 Stephane Lescuyer. *Formalizing and implementing a reflexive tactic for automated deduction in Coq*. PhD thesis, Université Paris Sud-Paris XI, 2011.

13 Filip Marić. Formal verification of a modern sat solver by shallow embedding into isabelle/hol. *Theoretical Computer Science*, 411(50):4333–4356, 2010. URL: https://www.sciencedirect.com/science/article/pii/S0304397510004937, doi:10.1016/j.tcs.2010.09.014.

14 Joao Marques-Silva and Ines Lynce. *SAT Solvers*, page 331–349. Cambridge University Press, 2014.

15 Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001. doi:10.1145/378239.379017.

16 Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

17 Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

18 Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 16, 2010.

19 Sadigh. Lecture 17: Logic ii. CS221, 2018. Spring 2018 lecture notes. URL: https://web.stanford.edu/class/archive/cs/cs221/cs221.1186/lectures/logic2.pdf.

20 Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

21 Arnoud van der Leer, Kobe Wullaert, and Benedikt Ahrens. Scott's representation theorem and the univalent karoubi envelope, 2025. URL: https://arxiv.org/abs/2506.22196, arXiv:2506.22196.