Formal Software Verification Project Report
SAT Solver

Nathan Schmidt

Aarhus University, Aarhus, Denmark
Student number: 202400801

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

1 Introduction

The *Boolean Satisfiability Problem (SAT)* consists in determining whether, given a formula of propositional logic, there exists a *valuation* that satisfies it, i.e., a mapping from the unknowns of the formula to boolean values such that the formula holds. Great numbers of problems in computer science can easily be reduced to SAT, and even though SAT is known to be $\mathcal{NP}$-complete by the Cook-Levin theorem of 1971 [1] and its many proofs since then, there exist many efficient *SAT-Solvers* for certain classes of formulas.

In this project, we implement a small formally verified SAT-Solver in the Coq proof assistant for formulas containing conjunctions, disjunctions, implications, and negations. To that end, we start by formalizing the syntax of such formulas. Then, we introduce their semantic interpretation given a specified valuation. Following the implementation of a syntactic optimizer that we show to simplify a formula to its minimal form, we conclude this work by implementing the actual solver based on a brute-force search algorithm and prove it to be both correct and complete.

2 Implementation and Encountered Problems

In this section, we present our implementation choices and compare them to discarded alternatives to justify them. We also describe what difficulties we encountered during development and how we addressed them.

2.1 Syntax

First, we formalize the abstract syntax of boolean formulas containing conjunctions, disjunctions, implications, and negations.

Abstract Syntax. We state an inductive definition of the type `form` with a set of constructors describing how to build up a formula:

$$p, q ::= x \mid \texttt{true} \mid \texttt{false} \mid p \wedge q \mid p \vee q \mid p \to q \mid \neg p \tag{1}$$

For identifiers, we introduce the type `id` with a single constructor `Id` that wraps a string. We also define an equality function comparing two identifiers and returning `true` if and only if their contained strings are equal. To ease proof development in the rest of the project, we prove some basic lemmas about the function by case distinction, namely *reflexivity*, *equivalence with propositional equality*, and *decidability* of equality of identifiers.

Concrete Syntax. To make reading and writing formulas easier, we use Coq's `Notation`s system, as well as a `Coercion` from an identifier as a formula. The main difficulty lies in assigning precedence levels to the individual constructors that pay respect to the commonly assumed binding of operators. Given the *binds stronger* relation ($>$), that is:

$$\neg > \wedge > \vee > \to \tag{2}$$

For example, $x \vee \neg y \wedge z \to \texttt{false}$ is interpreted as $(x \vee ((\neg y) \wedge z) \to \texttt{false}$.

2.2 Semantics

Now that we can write a formula in Coq, we want to define its semantics, i.e., if we interpret it as `true` or `false`.

Valuations. To do so, we need to know which boolean values to replace all identifiers occurring in a formula with. Therefore, we define the type `valuation`. A valuation is a function that, being passed an identifier, returns `true` or `false`, or, in other words, a valuation is a total map from identifiers to booleans. Our implementation is analogous to the total map defined by Pierce et al. in [2].

Again, to simply writing and reading valuations, we introduce a Notation `x !-> b ;; v` to override a valuation `v` with a new value `b` for `x`. At first, we used just a single semicolon, but this causes conflicts with list notations.

We also specify some lemmas of [2] for later reasonings. Their proofs rely on the functional extensionality axiom, stating that two functions are equal if and only if their applications to all their possible arguments are equal. It is known to be compatible with Coq's logical core.

Interpreter. The `valuation` type allows us to define a recursive interpreter function. Applied to a valuation and a formula, it returns `true` if and only if the formula holds by traversing a formula bottom-up and pattern matching on it. All the necessary functions, such as `andb`, `orb`, and `negb` are already implemented in Coq. They even suffice to compute the result of an implication since $p \to q$ is known to be equivalent to $\neg p \vee q$.

2.3 Optimizer

2.4 A Subsection Sample

Please note that the first paragraph of a section or subsection is not indented. The first paragraph that follows a table, figure, equation etc. does not need an indent, either.

Subsequent paragraphs, however, are indented.

Sample Heading (Third Level) Only two levels of headings should be numbered. Lower level headings remain unnumbered; they are formatted as run-in headings.

Sample Heading (Fourth Level) The contribution should contain no more than four levels of headings. Table 1 gives a summary of all heading levels.

Table 1. Table captions should be placed above the tables.

Heading level	Example	Font size and style
Title (centered)	**Lecture Notes**	14 point, bold
1st-level heading	**1 Introduction**	12 point, bold
2nd-level heading	**2.1 Printing Area**	10 point, bold
3rd-level heading	**Run-in Heading in Bold.** Text follows	10 point, bold
4th-level heading	*Lowest Level Heading.* Text follows	10 point, italic

Displayed equations are centered and set on a separate line.

$$x + y = z \tag{3}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead (see Fig. 1).

Theorem 1. *This is a sample theorem. The run-in heading is set in bold, while the following text appears in italics. Definitions, lemmas, propositions, and corollaries are styled the same way.*

Proof. Proofs, examples, and remarks have the initial word in italics, while the following text appears in normal font.

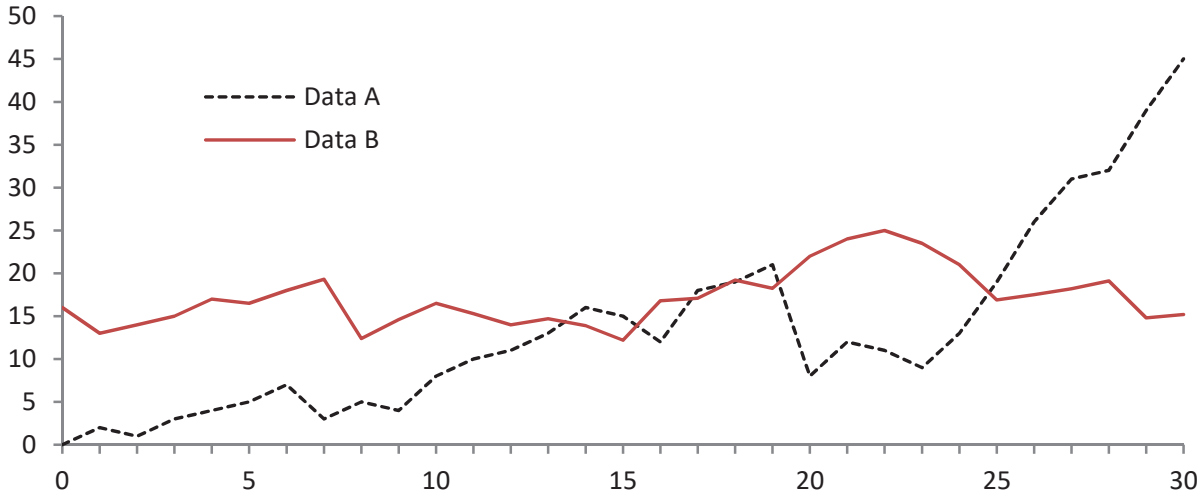

Fig. 1. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

For citations of references, we prefer the use of square brackets and consecutive numbers. Citations using labels or the author/year convention are also acceptable. The following bibliography provides a sample reference list with entries for journal articles [**ref_article1**], an LNCS chapter [**ref_lncs1**], a book [**ref_book1**], proceedings without editors [**ref_proc1**], and a homepage [**ref_url1**]. Multiple citations are grouped [**ref_article1**, **ref_lncs1**, **ref_book1**], [**ref_article1**, **ref_book1**, **ref_proc1**, **ref_url1**].

3 Conclusion

References

[1] S. A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047.

[2] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Logical Foundations*. Ed. by B. C. Pierce. Vol. 1. Software Foundations. Version 6.7, http://softwarefoundations.cis.upenn.edu. Electronic textbook, 2024.