RustSAT: A Library for SAT Solving in Rust

Christoph Jabs
University of Helsinki, Finland

Abstract

State-of-the-art Boolean satisfiability (SAT) solvers constitute a practical and competitive approach for solving various real-world problems. To encourage their widespread adoption, the relatively high barrier of entry following from the low level syntax of SAT and the expert knowledge required to achieve tight integration with SAT solvers should be further reduced. We present RustSAT, a library with the aim of making SAT solving technology readily available in the Rust programming language. RustSAT provides functionality for helping with generating (Max)SAT instances, writing them to, or reading them from files. Furthermore, RustSAT includes interfaces to various state-of-the-art SAT solvers available with a unified Rust API. Lastly, RustSAT implements several encodings for higher level constraints (at-most-one, cardinality, and pseudo-Boolean), which are also available via a C and Python API.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Rust, library, SAT solvers, constraint encodings

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.15

Supplementary Material
Software (RustSAT repository): `https://github.com/chrjabs/rustsat` [32]
archived at `swh:1:rel:ae4bbca3303bbd4d263bca42f01bf737638b36ce`
Software (Code & Data for Benchmarks): `https://github.com/chrjabs/rustsat-benchmarks` [33]
archived at `swh:1:dir:dd8bdc5aea20abd3ac594d6f24e58d1f8f05b480`

Funding This work is funded by Research Council of Finland (grant 356046).

Acknowledgements Thanks to the people who have already contributed to RustSAT on GitHub. Especially Christopher Jefferson, Jukka Suomela, and Noah Bruns have helped greatly in providing feedback based on different usecases for RustSAT. I also want to thank Matti Järvisalo and Jeremias Berg, who have supported me in working on RustSAT during my PhD research.

1 Introduction

Boolean satisfiability (SAT) solving is a significant success story of recent years [21]. State-of-the-art SAT solvers are highly optimized reasoning engines, able to solve propositional formulas with millions of variables and clauses, as demonstrated, e.g., by recent SAT Competitions [23, 28]. This has resulted in SAT solvers being commonly used as the underlying reasoning engine for solving a wide variety of problems, ranging from model checking [12, 9] over computer-assisted mathematics [50, 27, 51] to constraint satisfaction solving [44], and many more. Through their commonly supported incremental solving functionality [18, 38], SAT solvers also find application in problem-solving beyond NP [15, 6], and in optimization [3].

In order to achieve the level of performance optimization required, state-of-the-art SAT solvers are written in low-level languages, with C++ being the most common choice. This can pose some challenges when building tools based on SAT solving technology in different programming languages, as it often requires writing additional "boilerplate" code in order to interface with SAT solvers, which has to be repeated for each project and SAT solver. The standardized IPASIR interface [5] for incremental SAT solvers removes some of the required work, by unifying the API for the most-basic solver functionality. For the

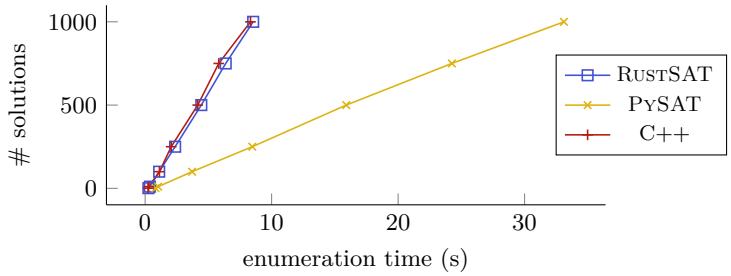

Figure 1 Runtime comparison of solution enumerator with RustSAT, PySAT, and C++ for the `AProVE11-12.cnf` instance.

Python ecosystem, the PySAT library [29, 31] has improved the accessibility of SAT solving technology significantly by packaging interfaces to state-of-the-art SAT solvers with helpful functionality for modelling problems as propositional logic. For interacting with SAT solvers from Python, OptiLog [1] defines a standardized `iSAT` C++ interface which includes more functionality than IPASIR. However, as many languages (e.g., Rust) offer direct interaction only with C APIs, `iSAT` is not universally applicable.

We present RustSAT, a library for the Rust programming language, that includes functionality for building and working with SAT instances, interfaces to common state-of-the-art SAT solvers, and CNF encodings for higher-level constraints. RustSAT is geared towards performance-sensitive applications written in Rust, where either SAT solvers are employed as part of the application, or propositional encodings for problem settings are encoded and then written to file. Some applications where RustSAT is already being used are a product configuration tool for the automotive industry [14], explaining pen and paper puzzles [20] (originally based on PySAT, but now using RustSAT), the Scuttle multi-objective MaxSAT solver [34], all written in Rust, and the Loandra [7] and MallobMax [49] MaxSAT solvers using RustSAT through its C API.

Being implemented in Rust, RustSAT provides a balance between accessibility – through Rust's modern build system and package distribution, as well as tooling to generate convenient API documentation – and performance. To illustrate the performance improvement that RustSAT can yield over PySAT in certain applications, as well as the little overhead incurred compared to not using any library at all, Figure 1 shows the time required to enumerate up-to 1000 solutions to the `AProVE11-12.cnf` instance from the SAT Competition 2022 Anniversary track, using PySAT, RustSAT and a custom C++ implementation.[1] While Python and PySAT can in many cases be competitive [29, 30], the more than 3× speedup observed in Figure 1 shows the potential efficiency improvement from implementing problem-solving tools using SAT solvers in a more efficient programming language such as Rust. Figure 1 also shows that RustSAT incurrs virtually no overhead compared to implementing solution enumeration in C++, employing the SAT solver directly. Additionally, Rust provides the advantages of type and memory safety, detecting many failure cases at compile time rather than at runtime.

All source code for RustSAT can be found at `https://github.com/chrjabs/rustsat`, and detailed and up-to-date API documentation is available at `https://docs.rs/rustsat`. Throughout this paper, when we refer to Rust types, functions or crates, we include hyperlinks to the API documentation online, formatted as in this example: `rustsat`.

[1] Experiment run on an AMD Ryzen 7 1700 (3 GHz) with 16 GB of memory, using the `pysat.examples.models` implementation included in PySAT (1.8.dev17) and the `enumerator` tool from `rustsat-tools`. All three approaches use MiniSat 2.2.0 as the SAT solver.

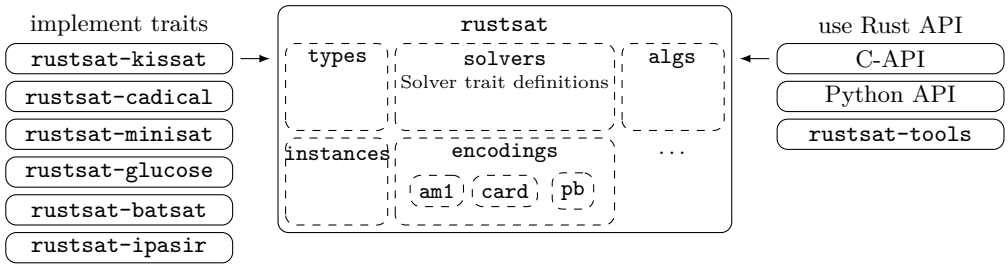

Figure 2 The architecture of the RustSAT project. Crates are represented as solid boxes, and modules as dashed boxes.

2 The Rust Programming Language

We give a brief overview of those features of the Rust programming language and its ecosystem that are relevant for understanding the design-principles of the RustSAT library.

Rust [52] is a programming language emphasizing performance, type safety, and reliability. In contrast to languages like Java or Python, Rust is fully compiled and does not use a garbage collector for memory management. In order to still achieve memory safety, Rust uses a system called the *borrow checker*, which during compilation tracks object lifetimes and ensures that all memory references are valid. For certain operations – e.g., when interfacing external C APIs or working with raw pointers – the Rust compiler cannot ensure memory safety. To enable usecases where such operations are necessary, Rust allows users to write so-called *unsafe* code, where memory safety guarantees need to be manually upheld by the user, as, e.g., when programming in C++.

Code inside a Rust project is typically structured into several (sub)modules, with paths to a type or function in a submodule being separated by two colons (`::`), a notation we also use throughout this paper. To enable generic code written against any type that supports specific functionality, Rust supports abstraction via so-called *traits*. Traits work similarly to interfaces in, e.g., Java: When defining a trait, one abstractly specifies certain methods that need to be provided by implementors of the trait. For example, a trait for SAT solvers might specify an `add_clause` and a `solve` method. A type that implements the trait then supplies the concrete implementations of the methods. Code that is generically written against a trait rather than a specific type can then be compiled with any type implementing that respective trait.

An important part of the Rust ecosystem is the `cargo` package manager. Similarly to the Python package manager `pip`, `cargo` takes care of downloading and compiling dependencies. Additionally, `cargo` provides a build system for projects, similar to `cmake` or GNU Autotools for C++ projects. A `cargo` package is called a *crate*, and crates are typically published on `https://crates.io`. The API documentation for crates on `crates.io` is automatically generated and available at `https://docs.rs/`.

3 Design Principles and Functionality

The architecture of RustSAT is illustrated in Figure 2. RustSAT consists of the main `rustsat` crate, which contains most of the functionality provided. Interfaces to SAT solvers, as well as the C and Python APIs are provided as separate crates. Additionally, the `rustsat-tools` crate contains some helpful executable tools for working with SAT-related instances, e.g., a tool to verify that a given assignment is indeed a solution to a CNF formula.

The aim of the RustSAT project is to be universal, convenient, and reliable to use, while not sacrificing performance. Towards this goal, we use an extensive automated test suite that is extended whenever bugs are discovered, to avoid regressions as much as possible. For reliability, we follow the semantic versioning scheme, where during this quite early stage of development (marked by major version 0), breaking changes will be published as a minor version bump, while any other changes are released as a patch version. All of RustSAT's functionality is accessible without requiring unsafe code. The only unsafe elements in the RustSAT API are intended as slight performance improvements when certain guarantees are already checked externally, and do therefore not need to be verified by RustSAT (e.g., `Lit::new_unchecked`). RustSAT is licensed under the MIT license, and developed entirely open-source. User feedback and contributions have already been very valuable in designing the API to be convenient in a wide range of usecases.

In the following, we detail the main functionality included in RustSAT version 0.7.2 split into functionality for working with SAT (and related) instances, solver interfaces, and higher-level constraint encodings.

3.1 Working with Problem Instances

In the `types` module, RustSAT defines various types that form the basis of working with SAT instances in Rust, and of the typesafe API of RustSAT. The type `Var` wraps a 32-bit integer to represent propositional variables (starting from index 0), and `Lit` represents literals as the variable index shifted one bit to the left, and the last bit signaling negation, i.e., with the same memory representation as in MiniSat [17]. The methods `Lit::from_ipasir` and `Lit::to_ipasir` convert literals to the integer representation used in IPASIR [5] and the DIMACS CNF [25] format. The `Clause` type is a wrapper around a vector providing some helpful functionality for working with clauses. Whenever a clause is referenced, RustSAT accepts any slice of literals (i.e., contiguous sequence in memory), making the `Clause` type optional to use. To build instances of satisfiability or optimization problems, the `SatInstance` and `OptInstance` types are available in the `instances` module.[2] As their constraints, these instance types support clauses, cardinality, and pseudo-Boolean constraints. To convert them to conjunctive normal form (`Cnf`), the constraint encodings described in Section 3.3 can be used. Objectives (`Objective`) can be represented either on the basis of soft clauses or as a linear function over literals. Instance objects can also be crated from DIMACS CNF [25] and WCNF [8], as well as OPB [48] files via parser implementations included in RustSAT. As an illustration of the API, slightly simplified method signatures of the `SatInstance` type are shown in Listing 1.

Variable management in RustSAT is handled by so-called variable managers that implement the `ManageVars` trait. Most central to the `ManageVars` trait is the `new_var` method, which is used for obtaining a variable that has not been used so far. The simplest variable manager simply keeps track of the next free variable index (`BasicVarManager`), while more advanced variable managers can maintain a mapping between propositional variables and any Rust objects (`ObjectVarManager`).

[2] If compiled with an additional feature flag, a `MultiOptInstance` type is available for multi-objective optimization.

Listing 1 Excerpt of the `SatInstance` API.

```
impl SatInstance {
  // Adding constraints
  pub fn add_clause(&mut self, cl: Clause);
  pub fn add_card_constr(&mut self, card: CardConstraint);
  pub fn add_pb_constr(&mut self, pb PbConstraint);
  pub fn add_lit_impl_clause(&mut self, a: Lit, b: &[Lit]);
  // File parsing
  pub fn from_dimacs_path(path: &Path) -> Result<Self>;
  pub fn from_opb_path(path: &Path, opts: opb::Options) -> Result<Self>;
  // Converting to CNF
  pub fn into_cnf(self) -> (Cnf, VarManager);
  // ...
}
```

3.2 SAT Solver Interfaces

The vast majority of state-of-the-art SAT solvers offer APIs in either C or C++. While calling C APIs from Rust is possible, the APIs for solvers often differ, with the IPASIR [5] interface for incremental solver standardizing only the most-central functionality. Furthermore, tightly integrating the build system of each solver with Rust's `cargo` build system to make them accessible in Rust requires additional work. RustSAT standardizes interfaces for common functionality in SAT solvers via a set of traits that are defined in the `solvers` module of the main `rustsat` crate. In separate crates, RustSAT packages various state-of-the-art SAT solvers that can be easily integrated in the `cargo` build system. These solver interface crates implement the traits defined in the `rustsat` crate, which enables easily swapping out a given SAT solver for another one that provides the same required functionality.

We first describe the traits defined by RustSAT, which unify the solver interfaces. Afterwards, we list the solvers RustSAT currently provides interfaces for, and which functionality they support.

All traits capturing SAT solver functionality are defined and documented in the `solvers` module. The documentation of this module also gives an up-to-date overview of the SAT solvers that are supported. The most basic functionality that every SAT solver supports is captured in the `Solve` trait. This trait includes methods for adding clauses, solving without assumptions, and obtaining solutions. *Incremental* solving [18, 38] is supported via the `SolveIncremental` trait, which contains the `solve_assumps` and `core` methods for solving under assumptions and obtaining a core. An overview of the most important methods in the `Solve` and `SolveIncremental` traits is given in Listing 2. Additional traits provide functionality for terminating solvers via callbacks (`Terminate`, compare `ipasir_set_terminate` [5]) or an asynchronous interrupt signal (`Interrupt`), tracking learned clauses (`Learn`, compare `ipasir_set_learn` [5]), setting preferred phases for variables (`PhaseLit`), excluding variables from being eliminated by pre/inprocessing (`FreezeVar`), flipping literals in found solutions (`FlipLit`), propagating assumptions (`Propagate`), setting resource limits (`LimitConflicts`, `LimitDecisions`, `LimitPropagations`), and getting solver statistis (`SolveStats`, `GetInternalStats`).

RustSAT provides crates for the state-of-the-art SAT solvers Kissat [11], CaDiCaL [10], MiniSat [17], and Glucose [2]. The detailed solver versions can be seen in Table 1; for some solvers multiple versions are supported, with the concrete version being selected via feature flags at compilation time. All except the first are incremental solvers and therefore implement the `SolveIncremental` trait. CaDiCaL supports the most functionality, implementing all

Listing 2 Excerpt of the `Solve` and `SolveIncremental` traits.

```rust
pub trait Solve {
  fn signature(&self) -> &'static str;
  fn solve(&mut self) -> Result<SolverResult>;
  fn lit_val(&mut self, lit: Lit) -> Result<TernaryVal>;
  fn add_clause(&mut self, clause: Clause) -> Result<()>;
  // ...
}

pub trait SolveIncremental {
  fn solve_assumps(&mut self, assumps: &[Lit]) -> Result<SolverResult>;
  fn core(&mut self) -> Result<Vec<Lit>>;
}
```

mentioned traits except for `LimitPropagations`. Details on which additional traits each solver implements can be found in the online documentation of the solver crates. In addition to those well-known solvers, RustSAT also implements an interface to BatSat [16], which is a reimplementation of MiniSat in Rust. Choosing BatSat allows for projects to be pure Rust, and therefore, e.g., compile to web assembly.

In addition to solver interfaces usable "out of the box", via the `rustsat-ipasir` crate RustSAT allows for linking to any user-provided library which implements IPASIR [5]. While CaDiCaL does implement the IPASIR interface, using CaDiCaL through its dedicated interface gives access to more functionality: the `rustsat-ipasir` crate only implements the `Solve`, `SolveIncremental`, `Learn`, `Term`, and `SolveStats` traits. Lastly, the `ExternalSolver` type allows for calling an executable solver binary with DIMACS CNF input, and parsing the output written to the standard output interface, given that it follows the output specification of the SAT competition [26]. This can be convenient for using solvers such as Gimsatul [22] that are not intended to be used as libraries.

3.3 Constraint Encodings

In the `encodings` module, RustSAT implements CNF encodings for higher-level constraints. Beyond simple encodings (e.g., a literal implying a clause, `atomics::lit_impl_clause`), mainly intended to increase code readability, encodings for at-most-one, cardinality, and pseudo-Boolean constraints are available. Providing efficiently implemented constraint encodings in RustSAT allows users to employ state-of-the-art constraint encodings when solving real-world problems without having to go through the error-prone process of implementing complex encodings themselves. Interfaces to the different constraint encodings are unified via

Table 1 SAT solver interfaces available for RustSAT.

Solver	Versions	Crate	Incremental
Kissat [11]	3.0.0 – 4.0.2	`rustsat-kissat`	no
CaDiCaL [10]	1.5.0 – 2.1.3	`rustsat-cadical`	yes
MiniSat [17]	2.2.0	`rustsat-minisat`	yes
Glucose [2]	4.2.1	`rustsat-glucose`	yes
Batsat [16]	0.6.0	`rustsat-batsat`	yes
IPASIR	–	`rustsat-ipasir`	yes
Call solver binary	–	`ExternalSolver`	no

Table 2 The constraint encodings implemented in RustSAT. Bound types supported by a specific encoding are indicated as upper bounds ($\leq$) and lower bounds ($\geq$).

Constraint	Encoding	Bounds	Incrementality
Pseudo-Boolean	`GeneralizedTotalizer` [36]	$\leq$	yes
	`BinaryAdder` [54, 19]	$\geq, \leq$	yes
	`DynamicPolyWatchdog` [43]	$\leq$	only changing bounds
Cardinality	`Totalizer` [4, 39]	$\geq, \leq$	yes
At-most-one	`Pairwise` [47]	≤ 1	no
	`Ladder` [24]	≤ 1	no
	`Bitwise` [46]	≤ 1	no
	`Commander` [37]	≤ 1	no
	`Bimander` [41]	≤ 1	no

traits: for at-most-one constraint encodings, the `am1::Encode` trait captures all functionality, while for cardinality and pseudo-Boolean constraint encodings upper and lower bounding, as well as incremental and non-incremental use are split into separate traits in the `card` and `pb` submodules. With incremental use of encodings we refer to either adding additional input literals, or changing the enforced bound, both while reusing previously built parts of the encoding [39]. Table 2 lists the encodings that are currently implemented and which functionality they support. To the best of our knowledge, RustSAT is currently the only readily available constraint encoding library providing an implementation of the dynamic polynomial watchdog encoding [43], which is used in state-of-the-art MaxSAT solvers [42].

For all constraint encodings implemented in RustSAT, we aim to produce the smallest number of clauses required to enforce the constraint that is being encoded. Towards this end, we employ the *cone of influence* strategy [43] which removes clauses from the encoding which contain pure literals – i.e., literals that only appear in one polarity – either directly, or after having already removed other clauses. Removing these clauses preserves both the correctness and the propagation properties of the encoding.

To illustrate the effect that the cone of influence strategy has, in Figure 3, we compare the number of clauses in CNF encodings generated by RustSAT (0.7.2), PySAT (1.8.dev17, note that the pseudo-Boolean constraint encodings provided are reexported from PbLib [45] via the PyPBLib interface), and Open-WBO (2.1) [40]. For the totalizer cardinality encoding [4], Figure 3 (left) shows the number of clauses in a totalizer encoding produced for 300 input literals and an enforced upper bound ranging from 1 to 300. It can be observed that PySAT seems to be generating the clauses required for enforcing *any* bound at all times. Open-WBO omits clauses only required to define the semantics for output variables corresponding to higher values than the currently enforced bound. In RustSAT, we also omit encoding output variables that are lower than the required range, which results in smaller encodings for higher bounds. Figure 3 also shows the number of generated clauses for binary adder (middle) and generalized totalizer (right) pseudo-Boolean encodings generated for 100–600 input literals with random weights in $[1, 100]$ and an enforced bound of 300. Also for the pseudo-Boolean encodings, RustSAT produces the fewest clauses, with PbLib and Open-WBO producing both more than $3\times$ the number of clauses for the binary adder, while the difference for the generalized totalizer encoding is smaller. Note that neither PbLib nor PySAT implement the generalized totalizer encoding.

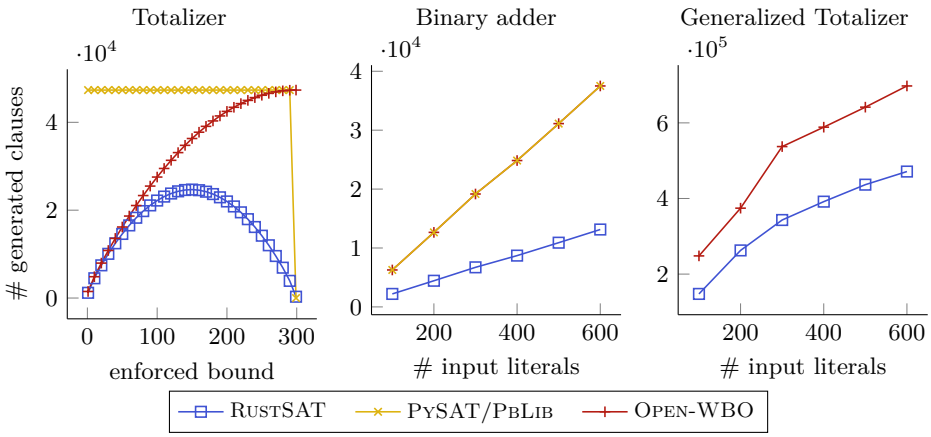

Figure 3 Left: Number of clauses in a totalizer cardinality encoding over 300 input literals for a given bound. Middle/Right: Number of clauses for a binary adder (middle) and generalized totalizer (right) pseudo-Boolean encoding for a given number of input literals with random weight in $[1, 100]$ and enforced bound of 300.

In addition to the CNF encodings listed in Table 2, in the `card::simulators` and `pb::simulators` submodules RustSAT provides helpers for inverting encodings (i.e., encoding an upper bound constraint with a lower-bounding encoding by inverting the constraint, and vice versa; `Inverted`), combining an upper and a lower-bounding encoding to get an encoding for both bound types at the same time, (`Double`) and encoding a pseudo-Boolean constraint by expanding it into a cardinality constraint with repeated input literals and using a cardinality encoding (`Card`).

Since the recent 0.7.0 release of RustSAT, the `Totalizer` and `GeneralizedTotalizer` encodings provide functionality for certifying the correctness of the generated CNF encoding by producing a proof in VeriPB format [13, 53, 35]. With this feature, the encodings can be employed in certified MaxSAT solvers, or for building a tool that produces certified translations from OPB to CNF. Currently, other encoding libraries such as PySAT or PbLib do not support generating certificates for the produced encodings.

3.4 C and Python API

While the primary intended way of using RustSAT is from the Rust programming language, some of its functionality is also exposed via a C and a Python API.

In version 0.7.2, the C API contains access for all higher-level constraint encodings listed in Table 2. In the C API, literals are represented as IPASIR-style `int`s, and clauses are returned via callbacks that work similarly to `ipasir_add` [5]. The main usecase for the C API is using the (incremental) encoding implementations in solvers written in a different language, as is for example the case for the Loandra MaxSAT solver [7]. To use the RustSAT C API in a project, the `rustsat-capi` crate needs to be compiled, which produces a statically linkable library. The full documentation of the C API can be found in `rustsat.h`.

Similarly to the C API, the Python API exposes all constraint encodings included in RustSAT. Additionally, the `Lit` and `Cnf` types, as well as a variable manager are included. At this point, the Python API of RustSAT does not include solver interfaces, but the RustSAT encodings can be used together with the solver interfaces in PySAT. The Python API is published on PyPI (`https://pypi.org/project/rustsat`) and its documentation is available online (`https://christophjabs.info/rustsat/pyapi`).

4 Conclusions

We presented RustSAT version 0.7.2, a library with the aim of making SAT solving technology more accessible from the Rust programming language. RustSAT packages tools for dealing with satisfiability and optimization instances in Rust, unified interfaces to state-of-the-art SAT solvers, and CNF encodings for at-most-one, cardinality, and pseudo-Boolean constraints. We have given an overview of the design principles of RustSAT– providing an easy-to-use API while not sacrificing performance – and illustrated key features empirically. RustSAT is available in open source, and detailed and up-to-date API documentation can be found online.

References

1 Josep Alos, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. OptiLog v2: Model, solve, tune and run. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2–5, 2022, Haifa, Israel.*, volume 236 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.25.

2 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11–17, 2009*, pages 399–404, 2009. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

3 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.

4 Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 – October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.

5 Tomas Balyo and Armin Biere. IPASIR: The standard interface for incremental satisfiability solving. https://github.com/biotomas/ipasir.

6 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021. doi:10.3233/FAIA201017.

7 Jeremias Berg, Christoph Jabs, Hannes Ihalainen, and Matti Järvisalo. Loandra in the 2024 MaxSAT evaluation. In Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian, editors, *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, volume B-2024-2 of *Department of Computer Science Report Series B*, pages 9–10. University of Helsinki, 2024.

8 Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian. MaxSAT Evaluation 2024: Rules. https://maxsat-evaluations.github.io/2024/rules.html.

9 Armin Biere. Bounded model checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 739–764. IOS Press, 2021. doi:10.3233/FAIA201002.

10 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification – 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

11 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.

12 Armin Biere and Daniel Kröning. SAT-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking.*, pages 277–303. Springer, 2018. doi:10.1007/978-3-319-10575-8_10.

13 Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. VeriPB and CakePB in the SAT Competition 2023. In Tomas Baylo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, volume B-2023-1 of *Department of Computer Science Report Series B*, pages 86–88. University of Helsinki, 2023.

14 Noah Frederik Bruns. Application of multi-objective MaxSAT-solvers for optimizing highly configurable products. Master's thesis, Technische Universität Wien, 2024.

15 Edmund M. Clarke. SAT-based counterexample guided abstraction refinement. In Dragan Bosnacki and Stefan Leue, editors, *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11–13, 2002, Proceedings*, volume 2318 of *Lecture Notes in Computer Science*, page 1. Springer, 2002. doi:10.1007/3-540-46017-9_1.

16 Simon Cruanes and Masaki Hara. BatSat: A (parametrized) rust sat solver originally based on minisat. https://github.com/c-cube/batsat.

17 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5–8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

18 Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In Ofer Strichman and Armin Biere, editors, *First International Workshop on Bounded Model Checking, BMC@CAV 2003, Boulder, Colorado, USA, July 13, 2003*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560. Elsevier, 2003. doi:10.1016/S1571-0661(05)82542-3.

19 Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2:1–26, 2006. doi:10.3233/sat190014.

20 Joan Espasa, Ian P. Gent, Ruth Hoffmann, Christopher Jefferson, and Alice M. Lynch. Using small MUSes to explain how to solve pen and paper puzzles. *CoRR*, abs/2104.15040, 2021. arXiv:2104.15040.

21 Johannes Klaus Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of SAT. *Commun. ACM*, 66:64–72, 2023. doi:10.1145/3560469.

22 Mathias Fleury and Armin Biere. Scalable proof producing multi-threaded SAT solving with gimsatul through sharing instead of copying clauses. *CoRR*, abs/2207.13577, 2022. doi:10.48550/arXiv.2207.13577.

23 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artif. Intell.*, 301:103572, 2021. doi:10.1016/j.artint.2021.103572.

24 Ian P Gent and Peter Nightingale. A new encoding of alldifferent into sat. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, volume 3, pages 95–110, 2004.

25 Maijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2024: Benchmarks. https://satcompetition.github.io/2024/benchmarks.html.

26 Maijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition output format. https://satcompetition.github.io/2024/output.html.

27 Marijn J. H. Heule and Manfred Scheucher. Happy ending: An empty hexagon in every set of 30 points. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 61–80. Springer, 2024. doi:10.1007/978-3-031-57246-3_5.

28 Marijn J. K. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *SAT Competition 2024*, volume B-2024-1 of *Department of Computer Science Report Series B*. University of Helsinki, 2024. URL: http://hdl.handle.net/10138/584822.

29 Alexey Ignatiev, António Morgado, and João Marques-Silva. PySAT: A python toolkit for prototyping with SAT oracles. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018. doi:10.1007/978-3-319-94144-8_26.

30 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient MaxSAT solver. *J. Satisf. Boolean Model. Comput.*, 11:53–64, 2019. doi:10.3233/SAT190116.

31 Alexey Ignatiev, Zi Li Tan, and Christos Karamanos. Towards universally accessible SAT technology. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21–24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 16:1–16:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.16.

32 Christoph Jabs. RustSAT. Software, version 0.7.2., swhId: swh:1:rel:ae4bbca3303bbd4d263bca42f01bf737638b36ce (visited on 2025-07-22). URL: https://github.com/chrjabs/rustsat, doi:10.4230/artifacts.24081.

33 Christoph Jabs. RustSAT Benchmarks. Software, swhId: swh:1:dir:dd8bdc5aea20abd3ac594d6f24e58d1f8f05b480 (visited on 2025-07-22). URL: https://github.com/chrjabs/rustsat-benchmarks, doi:10.4230/artifacts.24082.

34 Christoph Jabs. Scuttle: A multi-objective MaxSAT solver. https://bitbucket.org/coreo-group/scuttle.

35 Christoph Jabs, Jeremias Berg, Bart Bogaerts, and Matti Järvisalo. Certifying pareto optimality in multi-objective maximum satisfiability. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part II*, volume 15697 of *Lecture Notes in Computer Science*, pages 108–129. Springer, 2025. doi:10.1007/978-3-031-90653-4_6.

36 Saurabh Joshi, Ruben Martins, and Vasco Manquinho. Generalized totalizer encoding for pseudo-boolean constraints. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming – 21st International Conference, CP 2015, Cork, Ireland, August 31 – September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2015. doi:10.1007/978-3-319-23219-5_15.

37 William Klieber and Gihwon Kwon. Efficient CNF encoding for selecting 1 from n objects. In *Workshop on Constraints in Formal Verification*, 2007.

38 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

39 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming – 20th International Conference, CP 2014, Lyon, France, September 8–12, 2014.*

Proceedings, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014. doi:10.1007/978-3-319-10428-7_39.

40 Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver,. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014 – 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14–17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.

41 Van-Hau Nguyen and Son Thai Mai. A new method to encode the at-most-one constraint into SAT. In Huynh Quyet Thang, Le Anh Phuong, Luc De Raedt, Yves Deville, Marc Bui, Truong Thi Dieu Linh, Thi-Oanh Nguyen, Dinh Viet Sang, and Nguyen Ba Ngoc, editors, *Proceedings of the Sixth International Symposium on Information and Communication Technology, Hue City, Vietnam, December 3–4, 2015*, pages 46–53. ACM, 2015. doi:10.1145/2833258.2833293.

42 Tobias Paxian and Bernd Becker. Pacose: An iterative SAT-based MaxSAT solver. In Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian, editors, *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, volume B-2024-2 of *Department of Computer Science Report Series B*, page 26. University of Helsinki, 2024.

43 Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2018. doi:10.1007/978-3-319-94144-8_3.

44 Laurent Perron, Frédéric Didier, and Steven Gay. The CP-SAT-LP solver (invited talk). In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27–31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 3:1–3:2. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.3.

45 Tobias Philipp and Peter Steinke. PBLib – A library for encoding pseudo-boolean constraints into CNF. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015 – 18th International Conference, Austin, TX, USA, September 24–27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer, 2015. doi:10.1007/978-3-319-24318-4_2.

46 Steven D. Prestwich. Finding large cliques using sat local search. In Barry O'Sullivan Frédéric Benhamou, Narendra Jussien, editor, *Trends in Constraint Programming*, chapter 15, pages 269–274. John Wiley & Sons, Ltd, 2007. doi:10.1002/9780470612309.ch15.

47 Steven D. Prestwich. CNF encodings. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 75–100. IOS Press, 2021. doi:10.3233/FAIA200985.

48 Olivier Roussel. General opb format. https://www.cril.univ-artois.fr/PB24/OPBgeneral.pdf.

49 Dominik Schreiber, Christoph Jabs, and Jeremias Berg. From scalable SAT to MaxSAT: Massively parallel solution improving search. In Maxim Likhachev, Hana Rudová, and Enrico Scala, editors, *Eighteenth International Symposium on Combinatorial Search, SoCS 2025, Glasgow, UK, August 12–15, 2025*. AAAI Press, 2025. to appear.

50 Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is at least 14. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2–5, 2022, Haifa, Israel.*, volume 236 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.21.

51 Bernardo Subercaseaux, Wojciech Nawrocki, James Gallicchio, Cayden R. Codel, Mario Carneiro, and Marijn J. H. Heule. Formal verification of the empty hexagon number. In

Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9–14, 2024, Tbilisi, Georgia*, volume 309 of *LIPIcs*, pages 35:1–35:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ITP.2024.35.

52 The Rust Team. Rust: A language empowering everyone to build reliable and efficient software. https://www.rust-lang.org/.

53 Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning – 16th International Conference, LPNMR 2022, Genova, Italy, September 5–9, 2022, Proceedings*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, 2022. doi:10.1007/978-3-031-15707-3_33.

54 Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Inf. Process. Lett.*, 68:63–69, 1998. doi:10.1016/S0020-0190(98)00144-6.