Towards Universally Accessible SAT Technology

Alexey Ignatiev ✉ 🏠
Monash University, Melbourne, Australia

Zi Li Tan ✉
Monash University, Melbourne, Australia

Christos Karamanos ✉
Monash University, Melbourne, Australia

──── **Abstract** ────

Boolean satisfiability (SAT) solvers are a family of highly efficient reasoning engines, which are frequently used for solving a large and diverse variety of practical challenges. This applies to multidisciplinary problems belonging to the class NP but also those arising at higher levels of the polynomial hierarchy. Unfortunately, encoding a problem of user's interest to a (series of) propositional formula(s) in conjunctive normal form (CNF), let alone dealing with a SAT solver, is rarely a simple task even for an experienced SAT practitioner. This situation gets aggravated further when the user has little to no knowledge on the operation of the modern SAT solving technology. In 2018, the PySAT framework was proposed to address the issue of fast and "painless" prototyping with SAT solvers in Python allowing researchers to get SAT-based solutions to their problems without investing substantial time in the development process and yet sacrificing only a little in terms of performance. Since then, PySAT has proved a useful instrument for solving a wide range of practical problems and is now a critical package for the PyPI infrastructure. In the meantime, there have been advances in SAT solving and enhancements to PySAT functionality to extend its modelling and solving capabilities in order to make modern SAT technology accessible and deployable on a massive scale. This paper provides a high-level overview of the current architecture of PySAT and some of its capabilities including arbitrary Boolean formula manipulation, CNF preprocessing, and support for external user-defined propagators.

1 Introduction

Conflict-driven clause learning (CDCL) SAT solving serves as an illustrious example of a success story in Computer Science [32, 35, 36, 39, 58, 18, 19, 17, 42, 5, 4, 28, 9], providing a family of highly efficient decision oracles usable for solving myriads of practical problems. On the other hand, implementing solutions to practical problems based on the state-of-the-art SAT technology often requires one to be a SAT expert, which hampers the widespread use of SAT and its generalizations. Addressing the above issue was one of the motivations behind the proposal of the PySAT framework [24] designed specifically to ease incremental SAT-based prototyping. Since its inception, PySAT has become a valuable everyday instrument widely used in practice for tackling various AI problems. In 2021, based on the daily downloads statistics, PySAT has been included in the list of top-1% PyPI (Python Package Index) packages [48, 49] being named one of the critical projects for PyPI infrastructure.

Since the release of PySAT, there have been numerous enhancements made both in the framework and in the state of the art of SAT solving in general. Hence, the original paper [24]
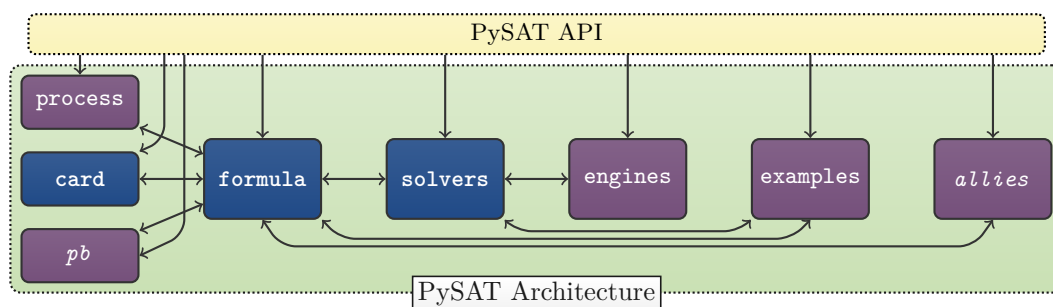
Figure 1 PySAT framework. Modules `solvers`, `formula`, and `card` shipped with the original version [24] appear in blue. Modules providing access to third-party tools appear in italics.

no longer serves as an up-to-date account of PySAT's capabilities. To fill in this gap, this paper reviews the current functionality of PySAT, focusing on its architecture and a few novel components, namely, arbitrary Boolean formula manipulation, CNF formula (pre-)processing and the capacity to accommodate external reasoning engines following the recent IPASIR-UP interface [20]. The paper also experiments with the latter demonstrating that it may be feasible and favorable to implement external propagators in Python as it appears to be a good trade-off between the development challenges and the overall solver performance.

2 Framework Architecture

Originally, PySAT comprised three core modules: `solvers`, `formula`, and `card` — providing access to state-of-the-art CDCL SAT solvers [34], CNF formula manipulation, and cardinality constraint encodings [50], respectively. Since then the list of modules expanded and now additionally includes modules `process` for formula processing (see Subsection 3.2), `examples` offering a variety of problem solving scripts serving to exemplify the use of PySAT, `engines` allowing a user to implement external propagators (see Subsection 3.3) as well as optional `pb` and `allies` modules interfacing with third-party libraries. While `pb` provides access to a list of pseudo-Boolean constraint encodings [50] by means of using the PyPBLib library [47, 41], `allies` is meant to offer direct access to external tools developed by the SAT community. Currently, a user can access the ApproxMCv4 approximate model counter [55, 54] and Unigen almost-uniform sampler [12, 11, 54] through the `allies` module. A simplified view on the current architecture of PySAT and its modules interconnection is shown in Figure 1.

3 Selected Novel Functionality

There have been numerous changes made in PySAT since its creation [24]. These include the support for additional SAT solvers, e.g., CaDiCaL 1.0.3, 1.5.3, and 1.9.5 [9] and MiniSat-based [18] solvers Glucose 4.2.1 [4, 22], MapleSAT [28, 31], MapleCM [29], MapleL-CMDistChronoBT [40], MergeSat 3.0 [30, 37], CryptoMiniSat [56], additional problem-solving tools like an award-winning MaxSAT solver RC2 [25], smallest minimal unsatisfiable subset (SMUS) extractor OptUx [27], and minimal hitting set enumerator Hitman [13]. As a single paper cannot encompass all the novel changes made, this section briefly overviews three selected components of the framework. (For a complete account of PySAT's capabilities, please refer to the online documentation.)

3.1 Arbitrary Boolean Formulas

In addition to the standard (W)CNF(+) functionality[1] of the original release of PySAT, the current version of the framework offers to users the ability to create *arbitrary* Boolean formulas and, importantly, to clausify them on demand through Tseitin transformation [57]. This may be of special importance to non-SAT researchers who are willing to use SAT for modeling and solving the problems from the scientific domains of their interests. To this end, PySAT exposes atomic expressions and various kinds of logic connectives (all inheriting from a base class `Formula`) as building blocks for creating complex Boolean formulas. For instance, variables can be created as atomic formulas, i.e., objects of the type `Atom`; they can be connected to one another using `And`, `Or`, `Neg`, and `Implies`, among a few other connectives.

▶ **Example 1.** A user may apply the bottom-up formula construction starting from variables, along these lines: `x, y, z = [Atom(c) for c in 'xyz']; f = ~(~x >> y) | (x & z)`, which will create a formula $f \triangleq \neg(\neg x \to y) \lor (x \land z)$.[2]

Furthermore, a user may employ `CNF` objects as components of larger formulas connected to the rest with any of the existing logic operators. To facilitate the use of arbitrary formulas, their on-the-fly clausification, integration of the `CNF` objects, as well as cardinality and pseudo-Boolean constraints handling, the framework offers a simple yet powerful manager of integer variable identifiers referred to as `IDPool`.

▶ **Example 2.** Given formula f in Example 1, calling `f.clausify()` will produce a list of clauses `[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4], [-3, 5]]`, which invokes `IDPool` to automatically assign integer identifiers `1`, `2`, and `4` to variables x, y, and z as well as identifiers `-3` and `5` for the two terms of the disjunction $\neg(\neg x \to y)$ and $x \land z$, respectively.

3.2 CNF Processing

CNF formulas can be (pre-)processed producing *equisatisfiable* CNF formulas using the facilities of PySAT's module `process`. This includes running some of the well-known *preprocessing* techniques such as bounded variable elimination, blocked clause elimination, failed literal probing, among many others [10]. A user may specify how many rounds of CNF processing should be applied as well as select the techniques to apply. In practice, applying such techniques may lead to formulas that are simpler to deal with than the original formulas. Furthermore, the power of formula processing may often suffice for proving unsatisfiability, without the need for a subsequent SAT call. Otherwise, assignments satisfying a processed formula can be mapped back to the original formula. Note that this module is implemented by exposing the preprocessing functionality offered by the CaDiCaL SAT solver [9].

▶ **Example 3.** The status of the result formula produced by the processor can be used to indicate whether or not the processor determined the initial formula to be unsatisfiable. As an example, consider an unsatisfiable input CNF formula $(\neg x_1 \lor x_2) \land (\neg x_2 \lor x_3) \land (\neg x_1 \lor \neg x_3) \land (x_1)$. If fed with this formula, the processor returns a new (processed) formula object containing

---

[1] CNF and WCNF formulas in PySAT [24] are represented as lists of clauses, each being a list of literals such that a positive integer `i` acts as a literal x_i while the negation $\neg x_i$ is represented by `-i`. CNF+ and WCNF+ formulas can also contain native cardinality constraints *understandable* by solvers like MiniCard [38].

[2] Observe how operators `~`, `&`, `|`, and `>>` are overloaded to act as `Neg`, `And`, `Or`, and `Implies`, respectively.

an empty clause and whose status is set to false, signifying unsatisfiability of the original formula, as follows:

```
>>> from pysat.process import Processor
>>> proc = Processor(bootstrap_with=[[-1, 2], [-2, 3], [-1, -3], [1]])
>>> processed = proc.process()
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[[]], False # result contains an empty clause and is unsatisfiable
```

▶ **Example 4.** Recall that the `process` module ensures equisatisfiability of an input formula and result formula. Furthermore, a satisfying assignment for the original formula can be restored given a satisfying assignment for the processed one. Consider a CNF formula $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1)$, which has a single model satisfying literals x_1 and $\neg x_2$. The processor constructs an empty (processed) formula and declares that the original formula is *not* determined to be unsatisfiable. Observe how one can get the unique assignment mentioned above restored from an assignment obtained for the processed formula:

```
>>> from pysat.process import Processor
>>> from pysat.solvers import Solver
>>> proc = Processor(bootstrap_with=[[-1, -2], [1, 2], [1]])
>>> processed = proc.process()
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[], True # result has no clauses and is not found to be unsatisfiable
>>> with Solver(bootstrap_with=processed) as solver:
...     st, mod = solver.solve(), solver.get_model()
...     print('status: {0}, model: {1}'.format(st, proc.restore(mod)))
status: True, model: [1, -2] # result is confirmed to be satisfiable
                             # and the correct model is restored
```

3.3 External Engines

Recent work [20] proposed an extension to the IPASIR interface [6] referred to as IPASIR-UP and added support for the new interface in CaDiCaL [9]. IPASIR-UP allows a user to specify an external propagator and attach it to a SAT solver supporting the interface. It has shown to be helpful in various practical scenarios when non-clausal reasoning is useful, e.g., in satisfiability modulo theories (SMT) solving [7, 8]. As a result, the interface is deemed highly valuable for extending applicability of SAT, especially if supported by additional solvers.

As handling low-level interaction between a SAT solver and an external propagation engine is a challenging and tedious task, PySAT takes on the mission to provide similar functionality directly in Python. Although using a reasoning engine written in a high-level programming language should arguably be slower than doing the same in a low-level language, this possibility aligns with the aims of PySAT to make the advanced SAT technology easy to use in practice.

The PySAT API for implementing external propagators offered by module `engines` is shown in Listing 1. (User-defined propagators should be defined as inheriting from the class `Propagator`.) Initially, the idea was to expose exactly the same interface as dictated by IPASIR-UP [20]. However, some of the methods in IPASIR-UP inherit the literal-by-literal interface of IPASIR, e.g., clauses are transferred through multiple calls to the same method, each passing a single literal. This overhead becomes noticeable if implemented in Python. Therefore, the interface is slightly modified in PySAT aiming to reduce the overhead of making repeated Python calls from C++ code. First, a clause is handed by the propagator to the solver as a single list of literals, which applies to methods `propagate()`, `provide_reason()`,

Listing 1 Interface for implementing external propagators in PySAT provided by module `engines`.

```python
class Propagator(object):
    def on_assignment(self, lit: int, fixed: bool = False) -> None:
        pass        # receive a new literal assigned by the solver

    def on_new_level(self) -> None:
        pass        # get notified about a new decision level

    def on_backtrack(self, to: int) -> None:
        pass        # process backtracking to a given level

    def check_model(self, model: list[int]) -> bool:
        pass        # check if a given assignment is indeed a model

    def decide(self) -> int:
        return 0  # make a decision and (if any) inform the solver

    def propagate(self) -> list[int]:
        return [] # propagate and return inferred literals (if any)

    def provide_reason(self, lit: int) -> list[int]:
        pass        # explain why a given literal was propagated

    def add_clause(self) -> list[int]:
        return [] # add an(y) external clause to the solver
```

and `add_clause()`. Internally, the C++ wrapper still follows the original interface by putting the literals into a queue. Second, PySAT gets rid of the *original* `has_external_clause()` check assuming that no clause is available if `add_clause()` returns `[]`. Finally, to avoid situations when a propagator does not communicate any useful information to the solver and runs for nothing, PySAT allows the propagator to disable itself on the fly (it is up to the author of a propagator to decide when it should happen) such that it will be invoked only when a SAT solver comes up with an assignment to be checked by `check_model()`. On-the-fly re-enabling of the propagator whenever it is favorable is also possible.

4   Distribution

The framework is distributed as an *open-source* project[3] with detailed installation instructions allowing a user to compile all the necessary and/or optional C++ components of the framework and get PySAT ready for use on their local machine. The easiest way to get PySAT is to install a pre-compiled binary wheel, i.e., Python package, from the PyPI repository [48] as follows:

```
$ pip install python-sat
```

Besides the source code distribution, the list of released binary wheels is quite extensive and contains 119 pre-built distributions targeting various versions of Linux, macOS, and Windows operating systems. Finally, PySAT is a part of the Pyodide project [46] whose aim

---

[3] `https://pysathq.github.io`

is to supply a version of Python compiled to WebAssembly and deliver a large collection of scientific computing packages available for execution *entirely in a web browser*. A nice side effect of this is that a user can implement their SAT-based solutions in Python and/or Javascript, to be operated in a browser, thus potentially expanding practical applicability of the SAT technology without the need to ever install PySAT.

## 5   Experimenting with External Engines

This section aims at showcasing the use of external propagator functionality offered in PySAT given two practical scenarios. We are essentially interested in testing how costly it is to run an external engine alongside CaDiCaL in practice and whether it defeats the purpose of implementing propagators in Python. With this in mind, we implemented an *example* propagator referred to as `BooleanEngine`, which should be general enough to attach various kinds of constraints on Boolean variables, including unweighted and weighted *linear* (i.e., cardinality and pseudo-Boolean, respectively) constraints or XOR-propagators, among other kinds of constraints. We implemented both weighted and unweighted linear constraints for the purpose of the experiment. Hereinafter, `BooleanEngine` reasoning on linear constraints is referred to as the *linear engine*. Here, we would like to remind the reader that the point is not to show that implementing a propagator in Python will necessarily outperform a low-level solution but rather to show that it may pay off in terms of the time invested in the development process, sacrificing little of the overall performance. Both experiments were run on a MacBook Pro running macOS Sonoma 14.3.1 with a 10-core Apple M1 Pro CPU and 32GByte RAM.

▶ Remark 5. The experimental results are presented in the form of cactus and scatter plots, e.g., see Figure 2a and Figure 2b, respectively. A cactus plot depicts multiple lines, each representing a particular competitor in terms of the statistic information on how many instances (on the X-axis) are successfully solved by this competitor within a given time limit (on the Y-axis). A scatter plot depicts in instance-by-instance comparison of two approaches by means of a set of points with coordinates (x, y), each representing a particular problem instance such that coordinate x signifies the time spent by one of the approaches (shown on the X-axis) dealing with this particular instance while coordinate y denotes the time spent on this instance by its competitor (shown on the Y-axis). Note that the green band in the scatter plots denotes the area where no approach outperforms the other by more than an order of magnitude.

5.1   Model Enumeration for Cardinality Constraints

Hereinafter, the first experiment is devoted to running the engine with cardinality constraints. In this case, we randomly generate 1000 systems of (unweighted) linear inequalities over 20 variables, i.e., each such inequality is of the form $\sum_{i=1}^{20} w_i \cdot l_i \leq v$, where $l_i \in \{x_i, \neg x_i\}$ such that $x_i \in \{0, 1\}$ and $w_i \in \{0, 1\}$, $v \in \{0, 1, \ldots, 20\}$. Inconsistent systems are filtered out, which results in 911 remaining problem instances. These remaining instances are either given to CaDiCaL augmented with the linear engine or given to pure CaDiCaL dealing with CNF formulas encoding the linear systems using cardinality networks [3].

Given that a solver may be lucky in finding a single satisfying assignment, the experiment is set to enumerate *all models* of the corresponding formulas. Depending on the formula, the number of models to enumerate varies from 1 to 1,044,905. Model enumeration is done by adding clauses blocking previously found models. For a fair comparison, the performance of a tool is measured as the overall time spent during the enumeration process, thus, ignoring

(a) Overall performance   **(b)** Instance-by-instance comparison
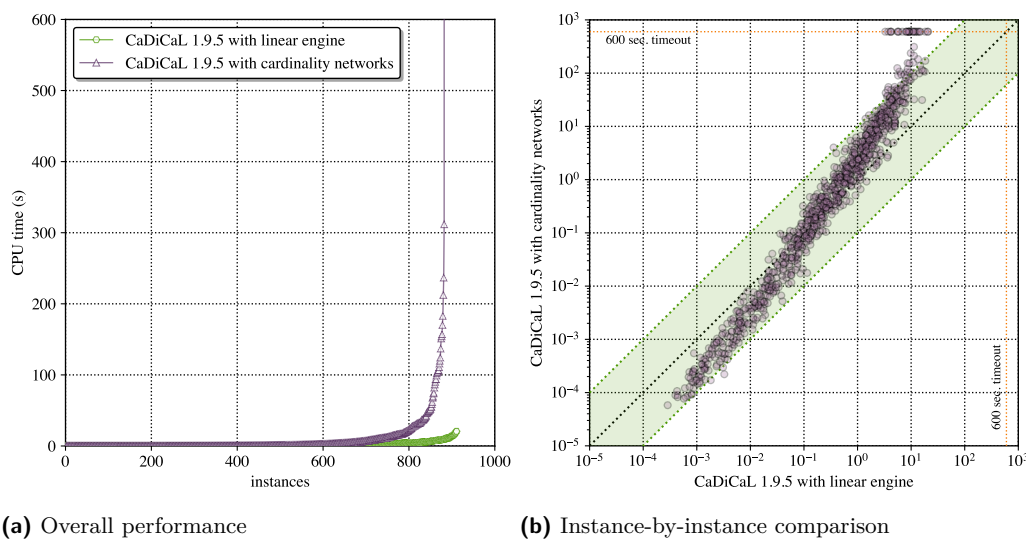
Figure 2 Linear engine vs cardinality networks.

the encoding time. The timeout value set for enumerating the models of a single formula is 10 minutes.

The performance of both competitors, i.e., CaDiCaL with and without the external linear propagator, is depicted in Figure 2. As can be observed, the configuration running the linear engine outperforms the competitor operating on CNF encodings of the inequality systems. In particular, the winner manages to solve all the instances spending *at most* 20.27 seconds per instance while the competitor on average spends much more time per instance and times out on 28 out of 911 instances.

5.2 Computing Formal Explanations for Tree Ensembles

The second experiment considers a more practical setting where we use CaDiCaL augmented with the external linear engine for computing and enumerating formal abductive explanations [52, 26, 33] for tree ensembles trained with the XGBoost algorithm [14]. Without diving into details, the task here is given a machine learning (ML) classification function $\kappa : \mathbb{F} \to \mathcal{K}$ mapping points in feature space $\mathbb{F}$ defined over $n = |\mathcal{F}|$ features $j \in \mathcal{F}$ to a class in $\mathcal{K}$ and a particular prediction $\kappa(\mathbf{v}) = c$, $\mathbf{v} \in \mathbb{F}$, and $c \in \mathcal{K}$, to compute a subset-minimal subset of features $\mathcal{X} \subseteq \mathcal{F}$ such that

$$\forall (\mathbf{x} \in \mathbb{F}). \left[\bigwedge\nolimits_{j \in \mathcal{X}} (x_j = v_j) \right] \to (\kappa(\mathbf{x}) = c)$$

Computing such a subset $\mathcal{X}$ requires one to make multiple calls to a reasoning oracle dealing with a logical representation of classifier κ. We reuse the propositional encoding of tree ensemble models proposed in [23] where feature domains and tree paths are CNF-encoded and each node in a tree is represented by a Boolean variable while a class weight $w \in \mathbb{R}$ assigned by a tree's terminal node t is modeled as a weighted soft clause (t, w). However, the use of the linear engine alongside CaDiCaL enables us to model the class selection process of boosted trees directly using pseudo-Boolean constraints, which compare the sums of weights for various classes, rather than by means of a MaxSAT objective function as in [23].

This experiment is twofold. First, it compares the performance of CaDiCaL augmented with the linear engine against the MaxSAT and SMT approaches [23] to computing a single

(a) Comparison against Z3

(b) Comparison against RC2
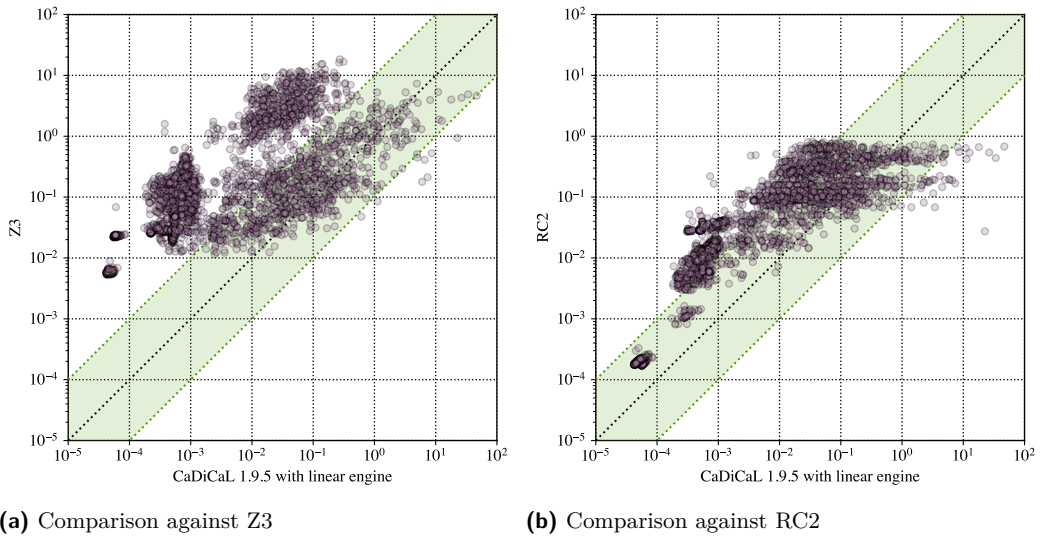
■ **Figure 3** CaDiCaL with linear engine vs MaxSAT and SMT on the task of explanation extraction.

explanation for the XGBoost models trained on a variety of publicly available datasets. (The SMT approach makes use of Z3 solver [16, 21] while the MaxSAT approach employs an optimized version of RC2 [25, 23].) The models trained comprise 50 trees per class, each of depth 1–5; the training (test, resp.) accuracy of these models is above 97% (85%, resp.). The experiment targets explaining 200 randomly selected instances from the corresponding datasets,[4] which resulted in 3755 individual problem instances. Second, it checks scalability of explanation *enumeration* with the MaxSAT vs CaDiCaL with external linear engine,[5] both set to enumerate 100 explanations for each of the models and instances considered above.

The results of single explanation computation is detailed in Figure 3. Observe that the slowest among the considered approaches is SMT while the fastest overall is MaxSAT. We should also mention that CaDiCaL with the linear engine is significantly less robust than MaxSAT as the time spent to extract an explanation with CaDiCaL varies much more. Finally, the use of the linear engine starts having performance drops if we increase models sizes; no such performance drops occur for MaxSAT or SMT.

The performance in explanation enumeration is detailed in Figure 4. Interestingly, this is where CaDiCaL with the external engine thrives and tends to outperform MaxSAT despite the fact that the engine is called hundreds of thousands of times per problem instance, which may be seen as surprising.

6   Related Work

While PySAT was originally inspired by PySMT's [21] capability of interfacing with various SMT solvers, there are Python APIs targeting *individual* SAT solvers, e.g., *PyMiniSolvers* [45] providing an API to MiniSat and MiniCard, *pycosat* [43] with the Python interface to PicoSAT, *satispy* [51] offering an API for MiniSat and lingeling, *pylgl* [44] for working with lingeling,

---

[4]   If a dataset has fewer than 200 instances, we explain each of the $n < 200$ *available* instances.

[5]   The implementation of [23] does not support explanation enumeration with SMT, which is why we do not compare against SMT in the explanation enumeration mode.
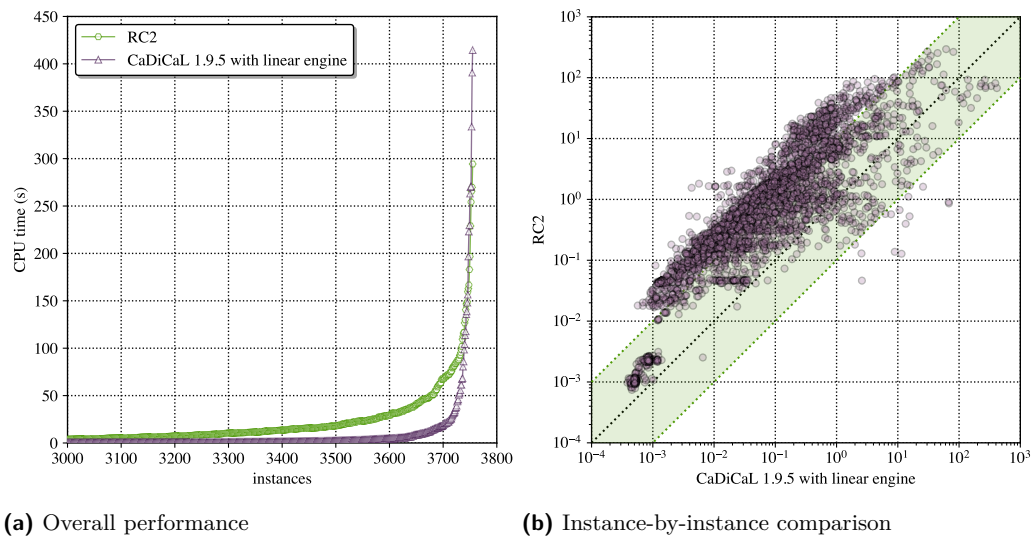
(a) Overall performance

(b) Instance-by-instance comparison

Figure 4 MaxSAT vs CaDiCaL with linear engine on the task of explanation enumeration.

and *pycryptosat* providing access to CryptoMiniSat [56, 53, 15]. Another framework called *OptiLog* [2, 1] offers a unified interface to multiple SAT solvers as well as access to cardinality and pseudo-Boolean constraint encodings. However, while OptiLog provides a user with a unified interface *iSAT* (through C++ and Python) to attach a SAT solver of their interest as well as with a way to configure, fine-tune, and benchmark it, PySAT's goal is different. Namely, PySAT is a fully open-source Python framework aiming to simplify prototyping with SAT oracles, even for researchers with little experience with SAT solving. It delivers a large range of solvers pre-installed accessible through the same API as well as a wealth of facilities to manipulate Boolean formulas and implement user-defined constraint reasoners.

7 Conclusions

It has been a number of years since the original release of the PySAT framework [24]. This paper provides a brief overview of its current capabilities. Besides a larger number of SAT solvers and problem-solving scripts integrated into the toolkit, these capabilities include facilities to manipulate arbitrary Boolean formulas and linear constraints as well as external reasoning engines by exploiting IPASIR-UP [20], and formula (pre-)processing and clausification [57]. PySAT is an easy-to-deploy and fully open-source Python package, whose mission is to make the advances of SAT universally accessible for solving problems arising in a wide range of scientific domains. Hopefully, the broader SAT community will assist this by contributing to PySAT with additional features and with proposals for further improvements.

References

1 Josep Alos, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. OptiLog V2: model, solve, tune and run. In *SAT*, pages 25:1–25:16, 2022.
2 Carlos Ansótegui, Jesus Ojeda, António Pacheco, Josep Pon, Josep M. Salvia, and Eduard Torres. OptiLog: A framework for SAT-based systems. In *SAT*, pages 1–10, 2021.
3 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *SAT*, pages 167–180, 2009.

4   Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving Glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *SAT*, pages 309–317, 2013.

5   Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.

6   Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016.

7   Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 1267–1329. IOS Press, 2021.

8   Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

9   Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1, pages 51–53, 2020.

10   Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In *Handbook of Satisfiability*, pages 391–435. IOS Press, 2021.

11   Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. On parallel scalable uniform SAT witness generation. In *TACAS*, pages 304–319, 2015.

12   Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *DAC*, pages 60:1–60:6, 2014.

13   Karthekeyan Chandrasekaran, Richard M. Karp, Erick Moreno-Centeno, and Santosh S. Vempala. Algorithms for implicit hitting set problems. In *SODA*, pages 614–629, 2011.

14   Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, pages 785–794, 2016.

15   CryptoMiniSat. `https://github.com/msoos/cryptominisat/`.

16   Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

17   Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.

18   Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

19   Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

20   Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: user propagators for CDCL. In *SAT*, pages 8:1–8:13, 2023.

21   Marco Gario and Andrea Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop*, 2015.

22   Glucose 3, Glucose 4.1, and Glucose 4.2.1. `http://www.labri.fr/perso/lsimon/glucose/`.

23   Alexey Ignatiev, Yacine Izza, Peter J. Stuckey, and João Marques-Silva. Using MaxSAT for efficient explanations of tree ensembles. In *AAAI*, pages 3776–3785, 2022.

24   Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

25   Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. RC2: an efficient MaxSAT solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019.

26   Alexey Ignatiev, Nina Narodytska, and Joao Marques-Silva. Abduction-based explanations for machine learning models. In *AAAI*, pages 1511–1519, 2019.

27   Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and Joao Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In *CP*, pages 173–182, 2015.

28   Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *SAT*, pages 123–140, 2016.

29   Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *IJCAI*, pages 703–711, 2017.

30   Norbert Manthey. The MergeSat solver. In *SAT*, pages 387–398, 2021.

31   MapleSAT. `https://maplesat.github.io/`.

32 Joao Marques-Silva. *Search algorithms for satisfiability problems in combinational switching circuits.* PhD thesis, University of Michigan, 1995.

33 João Marques-Silva and Alexey Ignatiev. Delivering trustworthy AI through formal XAI. In *AAAI*, pages 12342–12350, 2022.

34 Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 133–182. IOS Press, 2021.

35 Joao Marques-Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

36 Joao Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

37 MergeSat. `https://github.com/conp-solutions/mergesat`.

38 MiniCard 1.2. `https://github.com/liffiton/minicard/`.

39 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

40 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *SAT*, pages 111–121, 2018.

41 Tobias Philipp and Peter Steinke. PBLib - A library for encoding pseudo-Boolean constraints into CNF. In *SAT*, pages 9–16, 2015.

42 Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.

43 pycosat. `https://github.com/conda/pycosat`.

44 pylgl. `https://github.com/abfeldman/pylgl/`.

45 PyMiniSolvers. `https://github.com/liffiton/PyMiniSolvers/`.

46 Pyodide. `https://pyodide.org/`.

47 PyPBLib. `https://pypi.org/project/pypblib/`.

48 PyPI. `https://pypi.python.org/`.

49 PyPI 2FA Security Key Giveaway. `https://pypi.org/security-key-giveaway/`.

50 Olivier Roussel and Vasco M. Manquinho. Pseudo-Boolean and cardinality constraints. In *Handbook of Satisfiability*, pages 1087–1129. IOS Press, 2021.

51 satispy. `https://github.com/netom/satispy/`.

52 Andy Shih, Arthur Choi, and Adnan Darwiche. A symbolic approach to explaining Bayesian network classifiers. In *IJCAI*, pages 5103–5111, 2018.

53 Mate Soos. Enhanced gaussian elimination in dpll-based SAT solvers. In *POS@SAT*, pages 2–14, 2010.

54 Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *CAV*, pages 463–484, 2020.

55 Mate Soos and Kuldeep S. Meel. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *AAAI*, pages 1592–1599, 2019.

56 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, pages 244–257, 2009.

57 G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

58 Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.