Paramita: An Extensible Framework for SATisfiability Solving

Josep Alòs
Logic & Optimization Group (LOG), University of Lleida, Spain

Carlos Ansótegui
Logic & Optimization Group (LOG), University of Lleida, Spain

Juan Luis Esteban
Department of Computer Science, Universitat Politècnica de Catalunya - BarcelonaTech (UPC), Barcelona

Eduard Torres
Logic & Optimization Group (LOG), University of Lleida, Spain

Abstract

Paramita is a framework that provides a unified, plugin-based interface for interacting with external tools from the Satisfiability domain (e.g. SAT and MaxSAT solvers). These tools can be implemented in C++ or Python. The framework follows a strict separation between interface definitions and concrete implementations. Solvers and encoders in C++ are compiled as shared libraries and loaded at runtime. Solvers implementing IPASIR, IPASIR-UP and IPAMIR C interfaces can be directly integrated in Paramita with almost no effort. Paramita also supports a higher level modelling language for Non-CNF formulas, extendend with PB constraints and linear objective functions. Paramita is suitable for a wide range of combinatorial optimization and satisfiability tasks. This paper describes the architecture, solver and interfaces provided by the framework, the modelling language and other features.

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

Keywords and phrases Satisfiability, Maximum Satisfiability, Combinatorial Problems

1 Introduction

Satisfiability (SAT) solving and its optimization variant, Maximum Satisfiability (MaxSAT), form the algorithmic backbone of many modern constraint solving, planning, and combinatorial optimization applications. Over the past two decades, SAT solvers have matured enormously, and a prolific ecosystem of high-performance tools has emerged, driven in large part by annual competitions pushing the state of the art in both theory and engineering.

Despite this progress, integrating these tools into practical solving pipelines remains unnecessarily difficult. The diversity of solver APIs, input formats, and configuration interfaces creates significant friction for researchers and practitioners who wish to experiment with multiple solvers, compare their behaviour, or build portable applications that are not tied to a single implementation. Existing frameworks that attempt to unify solver access typically require developers to write substantial adapter code for each new tool, making the process of extending the framework a tedious and largely manual undertaking. Furthermore, framework developers are frequently required to assist in the integration of new solvers and extensions, creating a bottleneck that hinders the organic growth of the ecosystem and discourages independent contributions. As the community continues to produce new solving algorithms and specialised tools, the lack of a lightweight, standardised integration mechanism becomes an increasingly critical obstacle.

Paramita [1] addresses this problem by providing a *uniform plugin interface* for SAT-based tools. Rather than reimplementing any solving algorithm, it defines a set of abstract interfaces that any conforming solver, encoder, etc must satisfy. For example, concrete solvers are compiled independently as shared libraries (`.so` files on Linux/macOS, `.dll` on Windows) and loaded at runtime. This architecture means that Paramita itself remains lightweight and stable, while its ecosystem can evolve independently.

The rest of this paper is structured as follows. Section 2 discusses the related work. Section 3 gives an overview of the overall architecture and design principles. Section 4, 5, 6, 7, 8 shows how to work with fomulas, solvers, encoders, decoders and constraint propagators, respectively. Section 9 presents the modelling utilities in Paramita and Section 10 shows some experimental results.

2 Related Work

Paramita is a new framework for SATisfiability-based applications. It shares similarities with our previous framework OptiLog [6, 2]. We identified several areas for improvement in OptiLog. As a result, Paramita has been designed and implemented from scratch to provide a more powerful and robust framework.

To place Paramita framework in context, we present an overview of frameworks for solving combinatorial decision and optimization problems. Paramita can also be combined with several of these frameworks to address specific satisfiability-based applications. One possible taxonomy of these frameworks is the following:

General Constraint Programming (CP):

- SCIP [17] (`https://www.scipopt.org/`)
- CPMpy (`https://pypi.org/project/cpmpy/`)
- MiniZinc Python [56] (`https://python.minizinc.dev/`)
- Numberjack [32] (`https://pypi.org/project/Numberjack/`)
- EasyCP (`https://easycp.sourceforge.net`)
- PyCSP3 [42] (`https://pycsp.org/`)
- Savile Row (`https://github.com/savilerow`)

Mathematical Optimization / CP Hybrid Tools:

- COIN-OR [48] (`https://www.coin-or.org/`)
- Gurobi [31] (`https://www.gurobi.com/`)
- CPLEX [34] (`https://www.ibm.com/products/ilog-cplex-optimization-studio`)
- OR-Tools [63] (`https://developers.google.com/optimization`)
- CP-SAT [62] (`https://developers.google.com/optimization/cp/cp_solver`)
- Pyomo [20] (`https://www.pyomo.org/`)
- PuLP [53] (`https://pypi.org/project/PuLP/`)
- CVXPY [26] (`https://www.cvxpy.org/`)
- DOcplex [35] (`https://pypi.org/project/docplex/`)

SAT / SMT / Logic-Oriented:

- Z3 [24] (`https://github.com/Z3Prover/z3`)
- PyPbLib [64] (`https://pypi.org/project/pypblib/`)
- PySAT [36] (`https://pysathq.github.io/`)
- OptiLog [6, 2] (`https://ulog.udl.cat/static/doc/optilog/html/index.html`)

[1] `https://pypi.org/project/paramita/`

- PySMT [29] (`https://pypi.org/project/PySMT/`)
- PyNuSMV [19] (`https://pynusmv.readthedocs.io/`)
- clingo [30] (`https://potassco.org/clingo/`)
- PyEDA [27] (`https://pyeda.readthedocs.io/`)
- XCSP3 [18] (`https://xcsp.org`)

Combinatorial Planning / Scheduling:

- unified-planning [52] (`https://unified-planning.readthedocs.io`)
- OptaPy [21] (`https://pypi.org/project/optapy`)

Quantum Computing:

- D-Wave Ocean SDK (`https://docs.dwavequantum.com/en/latest/ocean/`)
- Qiskit SDK [41] (`https://qiskit-community.github.io/qiskit-optimization/`)

Paramita also supports several standard C interfaces for SAT and MaxSAT solvers (discussed in Section 5), namely IPASIR, IPASIR-UP, and IPAMIR:

- IPASIR is the reversed acronym of Re-entrant Incremental SAT Solver API. It is a standard C interface for incremental SAT solvers that allows applications to interact with SAT solvers through a common API by incrementally adding clauses, solving under assumptions, and reusing learned clauses. IPASIR is widely used in formal verification, model checking, and automated reasoning.
- IPASIR-UP extends IPASIR with support for user propagators and advanced interaction between applications and SAT solvers. It enables external control of propagation, conflict explanation, and tighter integration with domain-specific reasoning engines. This extension is useful for SMT solving, constraint programming, and hybrid reasoning systems.
- IPAMIR is a standard incremental interface for MaxSAT solvers. It extends the IPASIR paradigm to optimization problems involving hard and soft clauses. It supports incremental MaxSAT solving, weighted soft constraints, and optimization queries, making it suitable for scheduling, planning, and combinatorial optimization problems.

3 Paramita Architecture

Paramita's architecture revolves around the idea of plugins: external tools that can be loaded to the project. While Paramita itself is implemented using C++, through nanobind [39] we provide a Python wheel. Paramita is mainly structured around three key points:

- The interfaces that specify the contract between the project and the external tools. Along with each interface, we also provide a rich set of tests to ensure implementations conform to the specification required by Paramita.
- The Plugins implementation. To add an external tool to Paramita, we need the tool to implement the previous defined interfaces. Usually this is done with an adapter pattern, where the plugin implementation is a simple translation layer between existing methods in the tool and the methods required by Paramita. Once implemented, the plugin is compiled into a shared library ready to be loaded.
- The Core project containing all the necessary infrastructure for the plugins to be loaded, the enhancements built on top of the raw interfaces that the project provides and, the Python bindings to use Plugins from Python.

It is relevant to mention that the interfaces have a subset of the methods exposed as optional. This ensures that even tools with only a subset of the specified behaviour can be incorporated and used in the project.

3.1 Interfaces hierarchy

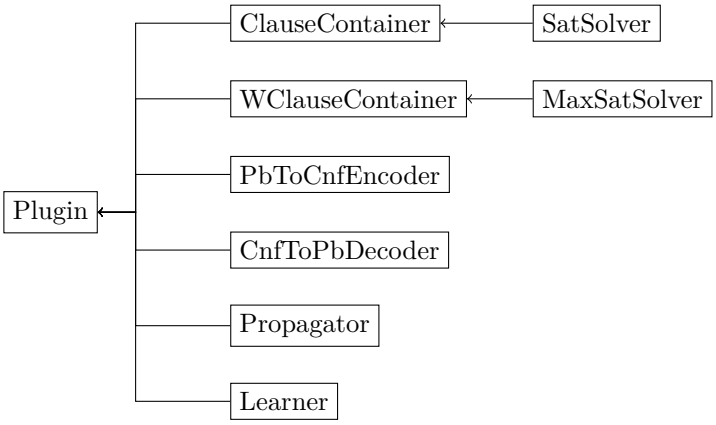


Figure 1 Interfaces hierarchy in Paramita.

Figure 1 shows the hierarchy of interfaces in Paramita. On top of the hierarchy we have the `Plugin` interface which provides methods to set and get additional attributes dynamically. These additional parameters can be used to define options or properties of the classes that inherit from Plugin interface or to retrieve additional data. This allows users to tailor the behaviour of generic tools for their specific use case.

▶ **Example 1.**

Inheriting from `Plugin` we have:

- `ClauseContainer`: this interface represents a class that can hold a set of clauses, and a clause hold a set of literals. We do not assume any particular operator on the set of clasues or literals. Therefore, this interface can support CNF, DNF, XOR formulas, etc.
- `WClauseContainer`: Similar to `ClauseContainer`, but introducing a weight to each clause in the container, with a special weight called `TOP_WEIGHT`.
- `PbToCnfEncoder`: Converts a pseudo-boolean expression to a CNF Boolean formula. They are described in more detail in Section 6.
- `CnfToPbDecoder`: Retrieves from a CNF Boolean formula a set of Pseudo Boolean constrains logically implied by the formula. They are described in more detail in Section 7.
- `Propagator`: Support for adding constraint propagators as described in [28]. They are explained in more detain in Section 8.
- `Learner`: Support for retrieving learned clauses in the solver as described in [14].

Following `ClauseContainer` we have `SatSolver`. Similarly, `MaxSatSolver` extends the idea of a `WClauseContainer`. Solvers are described in more detail in Section 5.

3.2 Plugins system

Developers can implement any of the Paramita interfaces in C++ (which then are compiled to a shared object library and dynamically loaded in Python) or directly in Python (by

sub-classing the desired interface). For the C++ implementations, Paramita automatically bridges C++ to Python, so from the end-user perspective both implementation options are the same. This is what we call Paramita's Plugin system.

This workflow allows any tool developer to start prototyping their tool in Python, and transition to a more efficient C++ version as they progress in the development. Alternatively, developers can implement a lightweight adapter wrapper around existing C/C++ tools to bring them to the Paramita framework.

Once Paramita has an implementation for any of the interfaces, their usage in the framework becomes completely transparent, being able to interact with any other part of Paramita, regardless if it is implemented in C/C++ or in Python.

This Plugin system decouples the Paramita framework development from the external tool ecosystem, as developers can add external tools without requiring a new release of the library (in contrast to what happens in PySAT [36]), and even can hot-load them in runtime (unlike in OptiLog [2], that despite do not requiring a new release of the framework as PySAT, it does need all the tools supported to be available upfront).

Finally, this architecture allows Paramita to be a lightweight package, with just 1.2 MB for the current Python wheel, as by default it does not include any external tool. In contrast, the last versions of OptiLog and PySAT are 4.2 MB and 3.1 MB respectively. This flexibility allows users to include in their applications only the external tools that they are using.

3.3 Core enhancements

The final key point of Paramita is the core, which enhances the capabilities of the tools implementing the raw interfaces to a full suite for SAT-based application developers.

Currently, the core provides the following:

- The python interface.
- Formulas representing CNF and WCNF. These formulas implement their respective container interface. They are explained in detail in Section 4.
- Loading of DIMACS CNF and WCNF (both original and revised format) directly into a `ClauseContainer` and a `WClauseContainer`, respectively. The loading supports both plain text files, as well as compressed ones (currently BZIP, XZ, GZIP, and ZSTD).
- Dumping CNF and WCNF formulas to files (with compression support).
- A rich modelling language. The problems represented in this language can be converted to CNF/WCNF using either a built-in translator, or with a user-defined one (suitable to specific use cases where developers want more control over the process). This is explained in detail in Section 9.
- Signal management on long running tasks for the solvers (solve, propagate...).
- Interoperability between Python and C++ implementations. For example, a user defines a custom propagator for a SAT solver in Python. While the SAT solver is implemented in C++, Paramita will adapt the Python class such that the SAT solver only sees a `Propagator` instance in C++.

4 Formulas

Paramita provides `CnfFormula` (inheriting from `ClauseContainer`) and `WcnfFormula` (inheriting from `WClauseContainer`) classes for storing and manipulating CNF and WCNF

formulas [2], respectively. These classes are exposed at the Python level, but are implemented in C++ for efficient formula manipulation.

▶ **Example 2.** These are some basic usage examples of `CnfFormula` and `WcnfFormula`:

```python
from paramita.containers import CnfFormula

cnf = CnfFormula()
cnf.add_clauses([[1, 2, 3], [-1, -2, -3]])
print(cnf.max_var())  # 3
print(cnf.num_clauses())  # 2
print(cnf.is_feasible([-1, -2, -3]))  # False
print(cnf.clauses())  # [[1,2,3],[-1,-2,-3]]

from paramita.containers import WcnfFormula

wcnf = WcnfFormula()
wcnf.add_clause([1, 2, 3])
wcnf.add_clause([-1], 1)
wcnf.add_clause([-2], 1)
wcnf.add_clause([-3], 1)
print(wcnf.max_var())  # 3
print(wcnf.num_clauses())  # {TOP_WEIGHT: 1, 1: 3}
print(wcnf.max_weight())  # 1
print(wcnf.total_weight())  # 3
print(wcnf.is_feasible([1, -2, -3]))  # True
print(wcnf.cost([1, -2, -3]))  # 1
print(wcnf.soft_clauses())  # [(1,[-1]),(1,[-2]),(1,[-3])]
print(wcnf.hard_clauses())  # [[1,2,3]]
```

Additionally, Paramita can efficiently load DIMACS CNF and WCNF formulas, in plain text, and in the *XZ*, *BZ2*, *GZIP* and *ZSTD* compressed formats. These formula loading methods support any `ClauseContainer` (or `WeightedClauseContainer`) implementation, so we can also use them to load the clauses directly to a `SatSolver` (or `MaxSatSolver`, see Section 5):

▶ **Example 3.**

```python
from paramita.containers import CnfFormula
from paramita.loaders import load_cnf

cnf = CnfFormula()
load_cnf("my-formula.cnf.gz", cnf)
print(cnf.num_clauses())
print(cnf.max_var())
```

Finally, users can implement their own versions of `CnfFormula` and `WcnfFormula` by extending the `ClauseContainer` and `WClauseContainer` interfaces. This can be useful to store clauses using specialized or custom data structures.

5 Solvers

From the perspective of the developer, currently, we can add SAT and MaxSAT solvers to Paramita [3]. The main interfaces to implement are the `SatSolver` interface for incremental SAT solvers and the `MaxSatSolver` interface for incremental MaxSAT solvers. The `SatSolver`

[2] Please, check appendix for definitions of CNF and WCNF.
[3] Please, check appendix for definitions of SAT and MaxSAT problems.

interface subsumes the C SAT interfaces IPASIR and IPASIR-UP and offers additional methods. Additionally, some of these methods are already implemented since can be defined in terms of other methods of the interface. With respect to MaxSAT, the `MaxSatSolver` interface subsumes the C MaxSAT interface IPAMIR. The developer can either implement the Python or the C++ versions (for efficiency or compatibility reasons) of `SatSolver` and `MaxSatSolver` interfaces.

Additionally, the developer can directly add a solver that already implements the IPASIR, IPASIR-UP, or IPAMIR interfaces, through the usage of the adapters `IpasirAdapter`, `IpasirUpAdapter` and `IpamirAdapter`, respectively. In particular, these adapters inherit from the `SatSolver` and `MaxSatSolver` interfaces. In this sense, it is easy to extend Paramita to accept other interfaces that may become standards in the community.

▶ **Example 4.** This example shows how a developer can add the Mergesat [49] SAT solver (version 4.0) to Paramita through the `IpasirAdapter` interface:

```
1  // solver.hpp
2  #pragma once
3
4  #include "minisat/core/ipasir.h"
5  #include "ipasir_adapter.hpp"
6
7  class Mergesat40 : public IpasirAdapter {
8  public:
9      Mergesat40 *clone() override;
10 };
11
12 // solver.cpp
13 #include <paramita/Exceptions.hpp>
14 #include "solver.hpp"
15
16 Mergesat40rc4 *Mergesat40::clone() {
17     throw UnsupportedMethodException("clone unsupported");
18 }
19
20 SATSOLVER_C_INTERFACE(Mergesat40);
```

Alternatively, we also provide the `NonIncrSatSolver` and `NonIncrMaxSatSolver` interfaces for non-incremental SAT solvers and MaxSAT solvers respectively, to differentiate these solvers through type checking instead of at runtime.

From the perspective of the user, currently we use Paramita within a Python environment (although we will release in the future a version that admits end-user applications in C++). That means that the user instantiates the Python `SatSolver` to interact with SAT solvers, and the Python `MaxSatSolver` to interact with MaxSAT solvers.

Tables 1 and 2 show the currently available SAT and MaxSAT solvers in Paramita. Please, check `https://ulog.udl.cat/static/doc/paramita` for the most up-to-date list of supported solvers.

6 Constraint Encoders

Currently, Paramita defines an interface to add and use encoders of PseudoBoolean Constraints into CNF called `PbToCnfEncoder`. This interface has a single method, `encode`. Here, we show the Python interface, although developers of encoders can also use the C++ interface.

Solver	URL	Versions
Kissat [15]	`https://github.com/arminbiere/kissat`	3.1.1, 4.0.1, SAT Competition 2024
Kissat-MAB-ESA [44]	`https://satcompetition.github.io/2024/index.html`	SAT Competition 2024
Kissat-MAB-DC [47]	`https://satcompetition.github.io/2024/index.html`	SAT Competition 2024
CominisatPS [61]	`https://bitbucket.org/coreo-group/ipamir/src/master/sat/cominisatps/COMiniSatPSChandrasekharDRUP.zip`	SAT Competition 2016
Glucose [7]	`https://github.com/audemard/glucose`	4.1, 4.2.1
Cadical [14]	`https://github.com/arminbiere/cadical`	1.9.5, 2.0.0, 2.1.3, 2.2.1, 3.0.0, SAT Competition 2025
Mergesat [50]	`https://github.com/conp-solutions/mergesat`	4.0-rc4
Maple SAT [46]	`https://maplesat.github.io/solvers`	73b36d3
Maple Glucose [45]	`https://maplesat.github.io/solvers`	73b36d3
Picosat [12]	`https://fmv.jku.at/picosat/`	9.6.5
Intel SAT [54]	`https://github.com/alexander-nadel/intel_sat_solver`	MaxSAT Evaluation 2024

Table 1 Currently supported SAT solvers and their versions.

Solver	URL	Versions
Loandra [11]	`https://github.com/jezberg/loandra`	b402e75
MaxCDCL [43]	`https://maxsat-evaluations.github.io/2024/mse24-solver-src/exact/unweighted/MaxCDCL2024.zip`	MaxSAT Evaluation 2024
WMaxCDCL [23]	`https://maxsat-evaluations.github.io/2024/mse24-solver-src/exact/weighted/WMaxCDCL2024.zip`	MaxSAT Evaluation 2024
OpenWBO [51]	`https://maxsat-evaluations.github.io/2024/mse24-solver-src/exact/unweighted/OpenWBO.zip`	MaxSAT Evaluation 2024
TT-OpenWBO-Inc [55]	`https://github.com/alexander-nadel-academic/tt-open-wbo-inc/`	5881d2d
UWrMaxSat [66, 8]	`https://github.com/marekpiotrow/UWrMaxSat`	1.8.0, MaxSAT Evaluation 2022
iMaxHS [58, 59]	`https://bitbucket.org/coreo-group/incremental-maxhs`	—
EvalMaxSAT2022 [8, 40]	`https://github.com/normal-account/EvalMaxSAT2022.git`	—
CGSS2 [38]	`https://bitbucket.org/coreo-group/cgss2/src/master/`	89c3b17

Table 2 Currently supported MaxSAT solvers and their versions.

These encoders can be incremental or not, where incrementality refers to the possibility of strengthening the bound on the PB constraint incrementally. Encoders are stateless, and the incrementality is handled through the `ContextState` received and returned by `encode`.

```python
class PbToCnfEncoder:

    def encode(self, pb: PbConstraint, container: ClauseContainer,
               context: Optional[ContextState]) -> ContextState:
        ...
```

Figure 2 Python interface of `PbToCnfEncoder`.

▶ **Example 5.** Lets focus on encoders using Sorting Networks for encoding Cardinality Constraints into CNF [10]. Roughly speaking, a Sorting Network is a circuit of n inputs $x_1, \ldots, x_n$ and n outputs $y_1, \ldots, y_n$, such that y_k is true if $\sum_{i=1}^{n} x_i \geq k$.

In Paramita, to encode the Cardinality Constraint $x_1 \vee \neg x_2 \vee x_3 \leq 2$ into CNF with a Non-Incremental Sorting Network we would execute the following code.

```python
from paramita import PbConstraint
from paramita.encoders import PbToCnfEncoder
from paramita.containers import CnfFormula

enc = PbToCnfEncoder.from_name("NonIncrSortingNetwork")
cnf = CnfFormula()
pb = PbConstraint(list(range(1, 101)), op=PbConstraint.LTE, bound=3)
enc.encode(pb, cnf)
print("After first encode:", cnf.num_clauses())
pb.bound = 2
enc.encode(pb, cnf)
print("After second encode:", cnf.num_clauses())
```

```
After first encode: 655
After second encode: 1181
```

For an incremental encoder, the usage would be:

```python
enc = PbToCnfEncoder.from_name("SortingNetwork")
cnf = CnfFormula()
pb = PbConstraint(list(range(1, 101)), op=PbConstraint.LTE, bound=3)
ctx = enc.encode(pb, cnf)
print("After first encode:", cnf.num_clauses())
pb.bound = 2
ctx = enc.encode(pb, cnf, ctx)
print("After second encode:", cnf.num_clauses())
```

Figure 3 Usage of incremental SortingNetwork encoder.

In contrast, this code produces the following output:

```
After first enforce: 655
After second enforce: 656
```

As observed, the only difference between non-incremental and incremental encoders is that we keep track on the encoder state through the context.

Currently, Paramita supports 20 non-incremental encoders and 4 incremental encoders from PBLib [65] (accessible at `https://github.com/master-keying/pblib`) and PySAT [37] (accessible at `https://github.com/pysathq/pysat/tree/master/cardenc`). For an up-to-date list, see the Paramita documentation.

7 Constraint Decoders

In the previous section, we described how Paramita can be used to encode Pseudo-Boolean (PB) constraints into CNF formulas. In general, these constraints correspond to the combinatorial structure explicitly modeled by the user when translating a problem into a SAT or MaxSAT instance. However, additional PB constraints may be implicitly present in the resulting formulation, that is, they may arise as logical consequences of the CNF formula without being explicitly identified by the user. Detecting and recovering such constraints can be beneficial for improving the efficiency of the solving process.

For instance, in SAT solving, it is possible to recover At-Most-One (AMO), At-Least-One (ALO) constraints -and its combination Exactly One constraints (EO)- [3] and exploit them through CSP-inspired heuristics [5]. Similarly, At-Least-K (ALK) constraints can be identified and re-encoded using more suitable CNF encodings [13, 68]. In the context of translating arbitrary PB constraints (not just Cardinality constraints), AMO and EO detection can be used also to produce more efficient encodigns into CNF of PB constraints [16, 4].

Currently Paramita provides an interface for integrating tools capable of detecting PB constraints of the form $\sum_{i=1}^{n} c_i \cdot l_i \diamond k$ implied by a SAT formula φ, where $L = \{l_1, \ldots, l_n\}$ is a set of literals over $vars(\varphi)$, $\diamond \in \{>, \geq, =, \neq, \leq, <\}$, and c_i and k are integer constants.

The interface in Paramita to add and use `CnfToPbDecoder` is as follows:

```
class CnfToPbDecoder:
    def decode(self, container: ClauseContainer,
               target_lits: list[int]) -> list[PbConstraint]:
        ...
```

Figure 4 Python interface of `CnfToPbDecoder`.

Where `container` keeps the clauses of the target CNF formula φ, `target_lits` denotes the set of literals L over which PB constraints implied by φ are to be detected. As `CnfToPbDecoder` inherits from `Plugin`, implementations can define other parameters such as the maximum number of detected pb constraints.

8 Propagators

As discussed in Section 5, the `SatSolver` interface subsumes the IPASIR-UP interface. This gives access to incorporate propagators. Propagators can be used, for example, when the direct translation of a Constraint into CNF leads to a more inefficient approach. Also, it provides a key piece to the connection to SMT or higher level Constraint Programming approaches. Additionally, propagators can be used to add meta algorithms on top of the SAT solver, as shown in Example 6.

▶ **Example 6.** In this example we show how to use propagators to apply a cube-and-conquer [33] approach on top of a SAT solver, generating the corresponding cubes.

```
class CubingEngine(Propagator):

    def __init__(self, depth: int, blocking_var: int) -> None:
        super().__init__()
        self.depth = depth
        self.blocking_var = blocking_var

        self.cubes_: list[list[int]] = []

```

```
        self._solver = None

        self.decision_level = 0
        self.trail: list[int] = []
        self.trlim: list[int] = []

    def setup_observe(self, solver):
        self._solver = solver
        # Observe all the variables
        vars = [i + 1 for i in range(self._solver.max_var())]
        for v in vars:
            self._solver.add_observed_var(v)
        self._solver.add_observed_var(self.blocking_var)

    def on_assignment(self, lits: list[int]) -> None:
        for lit in lits:
            if self._solver.is_decision(lit):
                # We only consider the literals that are decided
                self.trail.append(lit)

    def on_new_decision_level(self) -> None:
        self.decision_level += 1
        self.trlim.append(len(self.trail))

    def on_backtrack(self, to):
        # undoing the assignments
        while len(self.trlim) > to:
            while len(self.trail) > self.trlim[-1]:
                self.trail.pop()
            self.trlim.pop()

        # updating decision level
        self.decision_level = to

    def check_found_model(self, model) -> bool:
        return True

    def has_external_clause(self):
        return len(self.trail) >= self.depth

    def get_external_clause(self):
        if len(self.trail) >= self.depth:
            self.cubes_.append(list(self.trail))
            clause = [self.blocking_var] + [-lit for lit in self.trail]
            return clause
        return []
```

We can use this propagator as follows:

```
s = SatSolver.dynamic_load(sys.argv[1])
load_cnf(sys.argv[2], s)

blocking_var = s.max_var() + 1

prop = CubingEngine(depth=3, blocking_var=blocking_var)
s.connect_external_propagator(prop)
prop.setup_observe(s)

is_sat = s.solve([-blocking_var], SolvingBudget())
assert is_sat is False

print(f"Generated {len(prop.cubes_)} cubes")
for cube in prop.cubes_:
    print(cube)
```

We follow a similar interface for propagators than the one provided in PySAT [37] for CaDiCaL 1.9.5. Paramita currently supports propagators for all the CaDiCaL versions, thanks to the `CadicalAdapter` implementation that extends the `IpasirAdapter` (see Section 5). Adding a new CaDiCaL version supporting constraint propagators is as simple as inheriting from `CadicalAdapter`[4].

In Paramita, these propagators can be implemented in Python or C++ through `Propagator` interface.

9 Extended Pseudo-Boolean (EPB) Formulas

Paramita offers a modelling language for representing combinatorial decision and optimization problems. In this section, we demonstrate how Extended Pseudo-Boolean (EPB) formulas can be used to encode combinatorial decision problems, while multisets of weighted EPB formulas are employed to represent combinatorial optimization problems. Depending on the user specification, these formulas are eventually translated into SAT or MaxSAT instances.

▶ **Definition 7.** *We define a class of formulas, called* Extended Pseudo-Boolean (EPB) formulas*, where Boolean expressions and linear (pseudo-Boolean) constraints can be nested within each other. This formalism can be seen as a fragment of SMT [25, 9, 57], with pseudo-Boolean constraints as theory atoms and Boolean connectives providing propositional structure, allowing rich combinations of Boolean and linear 0–1 arithmetic reasoning. Let $\mathcal{V}$ be a set of Boolean variables. The set of formulas F is defined inductively by this grammar:*

$$F ::= x \mid \top \mid \bot \mid \neg F \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \to F_2) \mid (F_1 \leftrightarrow F_2) \mid \left(\sum_{i=1}^{n} c_i \cdot F_i \diamond k \right)$$

where $x \in \mathcal{V}$, $n \in \mathbb{N}$, each coefficient $c_i \in \mathbb{Z}$, $k \in \mathbb{Z}$, $\diamond \in \{>, \geq, =, \neq, \leq, <\}$, F_i are formulas, $\top$ is the constant true formula, always evaluating to 1, $\bot$ is the constant false formula, always evaluating to 0, and $\sum_{i=1}^{n} c_i \cdot F_i \diamond k$ is a PB inequality (constraint) (where each formula F_i is interpreted as a value in $\{0, 1\}$). In practice, the grammar also accepts $-F$ as an abbreviation of $-1 \cdot F$, and to express inequalities with the comparator in an arbitrary position.

Each formula F is evaluated under a valuation $v : \mathcal{V} \to \{0, 1\}$, extended recursively so that Boolean connectives have their usual semantics, and pseudo-Boolean (PB) inequalities are interpreted arithmetically over $\{0, 1\}$-valued subformulas.

A PB inequality is called nested *or* non-atomic *if it contains an operand $c_i \cdot F_i$ where F_i is a composite formula, that is, F_i is not a simple literal ($x, \neg x$) or constant ($\top, \bot$). Otherwise, the inequality is* atomic *or* non-nested*.*

A PB inequality is called top-level *or* non-inner *if it is not contained within any other inequality, i.e., it appears at the outermost level of the formula.*

An EPB formula is in Negation Normal Form (NNF) if negation is not allowed on top-level inequalities and for expressions that do not appear inside an inequality it is only allowed on Boolean variables.

▶ **Definition 8.** *The satisfiability problem for an EPB formula (SAT-EPB) asks whether there exists an assignment to the Boolean variables that the formula evaluates to true.*

[4] One can override the required methods in case there are changes in the CaDiCaL API. Even in this case, adding a new version of CaDiCaL through the `CadicalAdapter` should be easier than implementing the interface from scratch.

EPB formulas provide Paramita with a higher-level formalism than CNF, allowing more compact and user-friendly problem modelling while remaining amenable to efficient SAT and MaxSAT encodings. Additionally, this language can serve as an intermediate representation for translating richer formalisms into SAT and MaxSAT instances.

▶ **Definition 9.** *A* Weighted EPB (WEPB) Formula *is a pair* (w, F)*, where* F *is an EPB formula and* w*, its weight, is a natural number or* ∞*. The formula is* hard *if* $w = \infty$ *(or no weight is specified); otherwise, it is* soft*. Hard EPB formulas* (∞, F) *are also denoted as* $[F]$*.*

▶ **Definition 10.** *The Maximum Satisfiability problem for a multiset* ϕ *of Weighted EPB formulas (MaxSAT-WEPB) is to find an assignment in which the sum of weights of the falsified soft formulas is minimal, denoted by cost(*ϕ*), and all the hard formulas are satisfied.*

9.1 Translating (Weighted) EPB formulas to CNF (WCNF)

9.1.1 From EPB formulas to CNF Boolean formulas

An EPB formula φ can be translated into a quasi-equivalent [5] CNF Boolean formula Ω, as follows:

1. **Convert φ into Negation Normal Form (NNF)**: That involves: (1) Remove all the implications in φ. We traverse the formula in post-order, replacing $F_1 \leftrightarrow F_2$ for $(\neg F_1 \vee F_2) \wedge (\neg F_2 \vee F_1)$, and $F_1 \rightarrow F_2$ for $(\neg F_1 \vee F_2)$, and (2) traverse in pre-order, pushing down negations by applying the De Morgan's rule, Double Negation rule, and inverting the comparison operator (e.g. $\leq$ for $>$ and so on).
2. **Process top-level nested inequalities**: For every top-level nested PB inequality in φ, replace every non-atomic F_i in the inequality by a fresh auxiliary variable aux. Now, all arithmetic expressions in φ are atomic inequalities (i.e. PB constraints for which we have an encoder to translate them into CNF).
3. **Process atomic PB inequalities**: Replace every PB constraint in φ by a fresh auxiliary variable aux. Now, φ is a Boolean formula in NNF.
4. **Convert φ to CNF**: Add CNF(φ) to the output formula Ω.
5. **Encode atomic PB inequalities**: For each PB constraint τ and corresponding aux from step 3: (1) (if needed) Normalize PB constraint and (2) add CNF($aux \rightarrow PBtoCnfEncoder(\tau)$ and $\neg aux \rightarrow PBtoCnfEncoder(\neg\tau)$) to Ω .
6. **Encode subformulas F_i from step 2**: For each F_i and corresponding aux from step 2, apply recursively this procedure on $aux \rightarrow F_i$ and $\neg aux \rightarrow \neg F_i$.

Observations: Notice that, from a more general perspective, step 2 is applied whenever there is no encoder capable of translating the given expression directly into CNF. In such cases, we use the Tseitin transformation [69], replacing subformulas with auxiliary Boolean variables so that the resulting expression can be encoded into CNF using an available encoder.

Regarding the application of the Tseitin transformation, one may wonder why in steps 5 and 6 we do not simply introduce equivalences of the form $aux \leftrightarrow CNF(expr)$. This approach is correct only if $CNF(expr)$ does not introduce additional auxiliary variables. Otherwise, the implication $CNF(expr) \rightarrow aux$ is not sound. Indeed, it is possible to construct an interpretation that satisfies $expr$ while falsifying aux. This happens because the auxiliary

[5] See appendix for definition of quasi-equivalence.

variables introduced inside $CNF(expr)$ may be assigned values that falsify some clause in $CNF(expr)$, even though the original expression $expr$ itself remains satisfied.

Also, we can make the Tseitin transformation more efficient by only working on one direction of implications in steps 5 and 6 as in [67]. To do this, we check if variable aux appears in Ω with just one polarity. If the polarity is positive (negative) only work with $aux \rightarrow expr$ ($\neg aux \rightarrow \neg expr$).

If our *PBtoCnf* encoder requires a normalized PB constraint as input, we follow these steps: (1) convert the PB constraint into $\leq$ expression, (2) move non-constant terms to the Left Hand Side, (3) convert all negative terms into positive by replacing $-F$ for $-(1 - \neg F)$ and (4) move all constants to the Right Hand Side. See, for example, [60].

9.1.2 From multisets of WEPB formulas to WCNF Boolean formulas

Finally, a multiset of Weigthed EPB formulas ϕ can be translated into a MaxSAT equivalent [6] WCNF Boolean formula Ω, as follows:

1. For each Weighted EPB formula $(w, \varphi) \in \phi$:
   If $w = \infty$:
   a. For each clause $c \in CNF(\varphi)$, add hard clause $[c]$ to Ω.
   If $w \neq \infty$:
   a. Create a fresh auxiliary variable aux.
   b. Add soft clause (w, aux) to Ω.
   c. For each clause $c \in CNF(\varphi)$, add hard clause $[\neg aux, c]$ to Ω.

   Observation: Notice that $CNF(\varphi)$ corresponds to the procedure in section 9.1.1.

9.2 Translating EPB formulas with Paramita

To translate an EPB formula (modelling a combinatorial decision problem) into a quasi-equivalent CNF formula, Paramita provides the method `add_to_container`. This method receives the expression (EPB formula) to be translated, the container implementing the ClauseContainer interface where the clauses of the CNF representing the SAT instance will be added to, and a callable object responsible of transforming the EPB formula into CNF. It is also possible to use the `to_cnf` procedure provided natively in Paramita, or implement (in Python) a custom transformer to CNF to have a finer control of the process.

To translate a multiset of Weighted EPB formulas ϕ (modelling a combinatorial optimzation problem) into a MaxSAT equivalent WCNF formula, Paramita offers the function `add_to_weighted_container`. The main difference with respect the previous method is that it receives an *objective function* representing the linear aggregation of the soft Weighted EPB formulas in ϕ we want to maximally satisfy, an EPB formula representing the conjunction of the hard formulas in ϕ we must satisfy and any object implementing the WClauseContainer interface where the clauses of the resulting MaxSAT instance will be added to.

▶ **Example 11.** We provide an example for modelling and translating into CNF and WCNF.

```
1 from paramita.containers import CnfFormula
2 from paramita.modelling import Bool, DimacsVariablePool, \
3     to_cnf, add_to_container
4
```

[6] See appendix for definition of MaxSAT equivalence.

```
a, b, c, d, e = map(Bool, ("a", "b", "c", "d", "e"))

exp = (a + b <= 1) | ((c & d) + e >= 1)

cnf = CnfFormula()
pool = DimacsVariablePool()

add_to_container(exp, to_cnf, cnf, pool)
```

This generates the following CNF clauses:

$$(aux_1 \vee aux_2),\ (\neg aux_1 \vee \neg a \vee \neg b),\ (\neg aux_2 \vee aux_0 \vee e),\ (\neg aux_0 \vee c),\ (\neg aux_0 \vee d)$$

```
from paramita.containers import WCnfFormula
from paramita.modelling import Bool, DimacsVariablePool, \
    to_cnf, add_to_weighted_container

a, b, c, d, e = map(Bool, ("a", "b", "c", "d", "e"))

exp = (a + b <= 1) | ((c & d) + e >= 1)
obj = a + (b & c) * 2 + e

wcnf = WcnfFormula()
pool = DimacsVariablePool()

add_to_weighted_container(obj, exp, to_cnf, wcnf, pool)
```

This generates the following WCNF clauses:

$$(1, a),\ (2, aux_3),\ (1, e),\ [aux_1 \vee aux_2],\ [\neg aux_1 \vee \neg a \vee \neg b],\ [\neg aux_2 \vee aux_0 \vee e],$$
$$[\neg aux_0 \vee c],\ [\neg aux_0 \vee d],\ [\neg aux_3 \vee b],\ [\neg aux_3 \vee c]$$

10 Experimental Results

We conduct a series of experiments to demonstrate the effectiveness of Paramita. All experiments were executed on a computing cluster whose nodes are equipped with two AMD 7402 processors, each comprising 24 cores operating at 2.8 GHz, and 21 GB of RAM per core. We compare the performance of PySAT and Paramita (both Python and C++ versions) across several scenarios. You can find the detailed experimental evaluation in Section 3 of the Appendix.

In the first experiment, we measure the performance of translating PB instances from the Pseudo-Boolean Competition [1] into CNF. We use the same encoders both in Paramita and PySAT. For each instance we measure, how long it takes to parse the PB instance (this is the same code for both Paramita and PySAT), then we use a PBencoder to translate every PB constraints in the instances to CNF. Finally, we add the resulting CNF into the SAT solver Cadical 1.9.5. The average ratio of Paramita (Python version) versus PySAT, shows that Paramita is around 50% faster than PySAT.

In the second experiment, we evaluate the performance of generating cubes, as described in Section 8. We use the same algorithm for generating the cubes for both Paramita and PySAT. We selected the instances from the SAT competition 2025 [22]. For each instance, we measure how long it takes to apply the approach described in Section 8. We show that Paramita Python version is slightly faster than PySAT version, finishing some more instances, while Paramita C++ version is much faster than PySAT, finishing even more instances in the given time limit.

References

1 PB25 Pseudo-Boolean Competition 2025. Technical report, Glasgow, August 2025. URL: https://www.cril.univ-artois.fr/PB25/.

2 Josep Alòs, 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)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2022.25, doi:10.4230/LIPIcs.SAT.2022.25.

3 Carlos Ansótegui. *Complete SAT Solvers for Many-Valued CNF Formulas*. PhD thesis, University of Lleida, 2004.

4 Carlos Ansótegui, Miquel Bofill, Jordi Coll, Nguyen Dang, Juan Luis Esteban, Ian Miguel, Peter Nightingale, András Z Salamon, Josep Suy, and Mateu Villaret. Automatic detection of at-most-one and exactly-one relations for improved SAT encodings of pseudo-Boolean constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 20–36. Springer, 2019.

5 Carlos Ansótegui, Jose Larrubia, and Felip Manyà. Boosting chaff's performance by incorporating CSP heuristics. 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*, Lecture Notes in Computer Science, pages 96–107. Springer, 2003. doi:10.1007/978-3-540-45193-8_7.

6 Carlos Ansótegui, Jesús Ojeda, Antonio Pacheco, Josep Pon, Josep M. Salvia, and Eduard Torres. Optilog: A framework for sat-based systems. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 1–10, Cham, 2021. Springer International Publishing.

7 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

8 Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, Ruben Martins, and Andreas Niskanen, editors. *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Helsinki, Finland, 2022. MaxSAT Evaluation report. URL: http://hdl.handle.net/10138/347396.

9 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*, Frontiers in Artificial Intelligence and Applications, pages 1267–1329. IOS Press, 2021. doi:10.3233/FAIA201017.

10 K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery. doi:10.1145/1468075.1468121.

11 Jeremias Berg. The Loandra MaxSAT solver. Software, swhId: swh:1:dir:6babae3bfbb34e637ca99fab6b5f3a85f2263476 (visited on 2025-08-08). URL: https://github.com/jezberg/loandra/tree/boums-cadical-integration, doi:10.4230/artifacts.24096.

12 Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modelling and Computation*, 4(2-4):75–97, 2008.

13 Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In *Theory and Applications of Satisfiability Testing - SAT 2014*, pages 285–301, 2014.

14 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*, pages 133–152, Cham, 2024. Springer Nature Switzerland.

15 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. In {Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, Martin Suda}, editor, *Proc. of SAT Competition 2020 - Solver and Benchmark Descriptions*, volume vol. B-2020-1 of *Department of Computer Science Report Series B*, pages 50–53. University of Helsinki, 2020.

16 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. Compact MDDs for pseudo-boolean constraints with at-most-one relations in resource-constrained scheduling problems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 555–562, 2017. doi:10.24963/ijcai.2017/78.

17 Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Liding Xu. The scip optimization suite 9.0, 2024. URL: https://arxiv.org/abs/2402.17702, arXiv:2402.17702.

18 Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: An integrated format for benchmarking combinatorial constrained problems, 2024. URL: https://arxiv.org/abs/1611.03398, arXiv:1611.03398.

19 Simon Busard and Charles Pecheur. Pynusmv: Nusmv as a python library. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, pages 453–458, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38088-4_33.

20 Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, third edition, 2021. doi:10.1007/978-3-030-68928-5.

21 Christopher Chianelli and Geoffrey De Smet. Optapy: An ai constraint solver for python. URL: https://github.com/optapy/optapy.

22 Cayden Codel, Katalin Fazekas, Marijn J. H. Heule, Markus Iser, Cayden Codel, Katalin Fazekas, Marijn J. H. Heule, and Markus Iser. Proceedings of SAT Competition 2025 : Solver and Benchmark Descriptions. Technical report, TU Wien, August 2025. URL: https://repositum.tuwien.at/handle/20.500.12708/218424, doi:10.34726/10379.

23 Jordi Coll, Chu-Min Li, Shuolin Li, Djamal Habet, and Felip Manyà. Solving weighted maximum satisfiability with branch and bound and clause learning. *Computers & Operations Research*, 183:107195, 2025. URL: https://www.sciencedirect.com/science/article/pii/S0305054825002230, doi:10.1016/j.cor.2025.107195.

24 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

25 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

26 Steven Diamond and Stephen Boyd. Cvxpy: a python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.*, 17(1):2909–2913, January 2016.

27 Chris Drake. PyEDA: Python Electronic Design Automation, 2015. URL: https://github.com/cjdrake/pyeda.

28 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: User Propagators for CDCL. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:13, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2023.8, doi:10.4230/LIPIcs.SAT.2023.8.

29 Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 1–12, 2015.

30 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = asp + control: Preliminary report, 2014. URL: https://arxiv.org/abs/1405.3694, arXiv:1405.3694.

31 Gurobi Optimization, LLC, 2026. URL: https://www.gurobi.com.

32 Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan. Constraint programming and combinatorial optimisation in numberjack. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 181–185, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

33 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, pages 50–65, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

34 IBM. IBM ILOG CPLEX Optimization Studio, 2026. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio.

35 IBM. IBM® decision optimization CPLEX® modeling for python, 2026. URL: https://ibmdecisionoptimization.github.io/docplex-doc/.

36 Alexey Ignatiev, Antonio Morgado, and Joao 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*, pages 428–437, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94144-8_26.

37 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)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:11, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2024.16, doi:10.4230/LIPIcs.SAT.2024.16.

38 Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Refined Core Relaxation for Core-Guided MaxSAT Solving. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2021.28, doi:10.4230/LIPIcs.CP.2021.28.

39 Wenzel Jakob. nanobind: tiny and efficient c++/python bindings, 2022. https://github.com/wjakob/nanobind.

40 Matti Järvisalo, Jeremias Berg, Fahiem Bacchus, Ruben Martins, and Andreas Niskanen. Evalmaxsat 2022 repository, 2022. Repository for the 2022 MaxSAT Evaluation used in MaxSAT competitions. URL: https://github.com/normal-account/EvalMaxSAT2022.

41 Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with qiskit, 2024. URL: https://arxiv.org/abs/2405.08810, arXiv:2405.08810.

42 Christophe Lecoutre and Nicolas Szczepanski. Pycsp3: Modeling combinatorial constrained problems in python, 2024. URL: https://arxiv.org/abs/2009.00326, arXiv:2009.00326.

43 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Boosting branch-and-bound maxsat solvers with clause learning. *AI Communications*, 35(2):131–151, 2022. URL: https://journals.sagepub.com/doi/abs/10.3233/AIC-210178, arXiv:https://journals.sagepub.com/doi/pdf/10.3233/AIC-210178, doi:10.3233/AIC-210178.

44 S Li, CM Li, M Luo, J Coll, MS Cherif, D Habet, and F Manyà. Esa solvers, kissat mab binary and amsat in sat competition 2024. *Proceedings of SAT Competition 2024*, 2024.

45 Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. Mapleglucose and maplecms. *SAT COMPETITION 2016*, page 50, 2016.

46 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 123–140, Cham, 2016. Springer International Publishing.

47 Jinjin Liu, Jianmin Zhang, and Yan Sun. Kissat mab-dc in sat competition 2024. *SAT COMPETITION 2024*, page 20, 2024.

48 Robin Lougee. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47:57 – 66, 02 2003. doi:10.1147/rd.471.0057.

49 Norbert Manthey. The mergesat solver. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 387–398. Springer, 2021. URL: https://doi.org/10.1007/978-3-030-80223-3_27.

50 Norbert Manthey. The mergesat solver. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 387–398, Cham, 2021. Springer International Publishing. URL: https://doi.org/10.1007/978-3-030-80223-3_27.

51 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*, pages 438–445, Cham, 2014. Springer International Publishing.

52 Andrea Micheli, Arthur Bit-Monnot, Gabriele Röger, Enrico Scala, Alessandro Valentini, Luca Framba, Alberto Rovetta, Alessandro Trapasso, Luigi Bonassi, Alfonso Emilio Gerevini, Luca Iocchi, Felix Ingrand, Uwe Köckemann, Fabio Patrizi, Alessandro Saetti, Ivan Serina, and Sebastian Stock. Unified planning: Modeling, manipulating and solving ai planning problems in python. *SoftwareX*, 29:102012, 2025. URL: https://www.sciencedirect.com/science/article/pii/S2352711024003820, doi:10.1016/j.softx.2024.102012.

53 Stuart Mitchell, Michael O'Sullivan, and Iain Dunning. Pulp: A linear programming toolkit for python, 2011. URL: https://optimization-online.org/2011/09/3178/.

54 Alexander Nadel. Solving huge instances with intel(r) SAT solver. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, Alghero, Italy, July 4-8, 2023*, LIPIcs, pages 17:1–17:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.SAT.2023.17, doi:10.4230/LIPICS.SAT.2023.17.

55 Alexander Nadel. Tt-open-wbo-inc: An efficient anytime maxsat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 15:1–7, 09 2024. doi:10.3233/SAT-231504.

56 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74970-7_38.

57 Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006. doi:10.1145/1217856.1217859.

58 Andreas Niskanen, Jeremias Berg, and Matti Järvisalo. Enabling Incrementality in the Implicit Hitting Set Approach to MaxSAT Under Changing Weights. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 44:1–44:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2021.44, doi:10.4230/LIPIcs.CP.2021.44.

59 Andreas Niskanen, Jeremias Berg, and Matti Järvisalo. Incremental Maximum Satisfiability. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2022.14`, `doi:10.4230/LIPIcs.SAT.2022.14`.

60 Jakob Nordström. Pseudo-boolean solving and optimization. Tutorial at the Simons Institute for the Theory of Computing, February 2021. "Satisfiability: Theory, Practice, and Beyond" Boot Camp. URL: `https://jakobnordstrom.se/docs/presentations/TalkSimons2102_PseudoBooleanSolvingTutorial.pdf`.

61 Chanseok Oh. Cominisatps the chandrasekhar limit and ghackcomsps,". *SAT Competition*, 2016.

62 Laurent Perron and Frédéric Didier. CP-SAT. URL: `https://developers.google.com/optimization/cp/cp_solver/`.

63 Laurent Perron and Vincent Furnon. OR-Tools. URL: `https://developers.google.com/optimization/`.

64 Tobias Philipp and Peter Steinke. Pblib – a library for encoding pseudo-boolean constraints into cnf. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 9–16, Cham, 2015. Springer International Publishing.

65 Tobias Philipp and Peter Steinke. Pblib – a library for encoding pseudo-boolean constraints into cnf. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer International Publishing, 2015. `doi:10.1007/978-3-319-24318-4_2`.

66 Marek Piotrów. Uwrmaxsat: Efficient solver for maxsat and pseudo-boolean problems. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 132–136, 2020. `doi:10.1109/ICTAI50040.2020.00031`.

67 David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. URL: `https://www.sciencedirect.com/science/article/pii/S0747717186800281`, `doi:10.1016/S0747-7171(86)80028-1`.

68 Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 20-25, 2024, Proceedings, Part I*, volume 14707 of *Lecture Notes in Computer Science*, pages 110–132, 2024.

69 G. S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. `doi:10.1007/978-3-642-81955-1_28`.

(Appendix) Paramita: An Extensible Framework for SATisfiability Solving

Josep Alòs
Logic & Optimization Group (LOG), University of Lleida, Spain

Carlos Ansótegui
Logic & Optimization Group (LOG), University of Lleida, Spain

Juan Luis Esteban
Department of Computer Science, Universitat Politècnica de Catalunya - BarcelonaTech (UPC), Barcelona

Eduard Torres
Logic & Optimization Group (LOG), University of Lleida, Spain

1 Preliminaries

▶ **Definition 1.** *An arbitrary propositional formula is an expression built recursively from propositional variables x, logical connectives ($\neg$,$\wedge$,$\vee$,$\rightarrow$,$\leftrightarrow$) [1], and constants 0 or $\bot$ (False) and 1 or $\top$ (True). When using the term* constraint *we will also refer to an arbitrary propositional formula. Abusing the term, we will also refer to arbitrary propositional formulas as Non-CNF formulas in this context.*

▶ **Definition 2.** *A literal is a propositional variable x or a negated propositional variable $\neg x$. A clause is a disjunction of literals. An empty clause, denoted as $\square$, has no literals. A clause with just one literal is a unit clause. A Conjunctive Normal Form (CNF) is a conjunction of clauses. We will also refer to a propositional formula in CNF form as a SAT formula (or SAT instance). Abusing the notation, we can denote a CNF formula that contains an empty clause as $\square$ since we are using this notation as a replacement for the constant False.*

▶ **Definition 3.** *A weighted clause is a pair (c, w) (also noted as (w, c) or ${}_{(w)}\, c$), where c is a clause and w, its weight, is a natural number or infinity. A clause is hard if its weight is infinity (or no weight is given); otherwise, it is soft. We will denote also hard clauses (c, ∞) as $[c]$. A Weighted Partial MaxSAT instance is a multiset of weighted clauses. A formula is in Weighted Normal Form (WCNF) if it is a conjunction of weighted clauses.*

▶ **Definition 4.** *A truth assignment for an instance φ is a mapping that assigns to each propositional variable in φ either 0 (False) or 1 (True). A truth assignment is* partial *if the mapping is not defined for all the propositional variables in φ.*

▶ **Definition 5.** *A truth assignment I satisfies a literal x ($\neg x$) if I maps x to 1 (0); otherwise, it is falsified. A truth assignment I satisfies a clause if I satisfies at least one of its literals; otherwise, it is violated or falsified. The cost of a clause (c, w) under I is 0 if I satisfies the clause; otherwise, it is w. Any truth assignment satisfies (falsifies) constant True (False). Given a partial truth assignment I, a literal or a clause is undefined if it is neither satisfied nor falsified. A clause c is a unit clause under I if c is not satisfied by I and contains exactly one undefined literal. An empty clause, $\square$, can not be satisfied.*

▶ **Definition 6.** *An unsatisfiable core is a subset of clauses of a SAT instance that is unsatisfiable.*

---

[1] where $A \rightarrow B$ and $A \leftrightarrow B$ are abbreviations of $\neg A \vee B$ and $(A \rightarrow B) \wedge (B \rightarrow A)$, respectively

▶ **Definition 7.** *The cost of a formula ϕ under a truth assignment I, denoted by $cost(I, \phi)$, is the aggregated cost of all its clauses under I.*

▶ **Definition 8.** *The Weighted Partial MaxSAT problem for an instance ϕ is to find an assignment in which the sum of weights of the falsified soft clauses is minimal, denoted by $cost(\phi)$, and all the hard clauses are satisfied. The Partial MaxSAT problem is the Weighted Partial MaxSAT problem where all weights of soft clauses are equal. The SAT problem is the Partial MaxSAT problem when there are no soft clauses. An instance of Weighted Partial MaxSAT, or any of its variants, is unsatisfiable if its optimal cost is ∞. A SAT instance ϕ is satisfiable if there is a truth assignment I, called a model, such that $cost(I, \phi) = 0$.*

▶ **Definition 9** (Logical equivalence)**.** *φ_1 and φ_2 are* **logically equivalent***, written as $\varphi_1 \equiv \varphi_2$, iff $\varphi_1 \models \varphi_2$ and $\varphi_2 \models \varphi_1$, (i.e, have the same models).*

▶ **Definition 10** (Equi-satisfiability)**.** *: φ_1 and φ_2 are* **equi-satisfiable***, written as $\varphi_1 \approx \varphi_2$, if φ_1 is satisfiable iff φ_2 is satisfiable (not necessarily in the same models).*

▶ **Definition 11** (Equi-assignability)**. Equi-assignability (or vars(φ_1)-restricted equivalence)***: φ_1 and φ_2 are* **equi-assignable** *if each model of φ_1 can be extended to some (not necessarily unique) model of φ_2.*

▶ **Definition 12** (Equi-countability)**. Equi-countability***: φ_1 and φ_2 are* **equi-countable** *if both formulas have the same number of models.*

▶ **Definition 13** (Quasi-equivalence)**. Quasi-equivalence***: φ_1 and φ_2 are* **quasi-equivalent** *if both are equi-assignable and equi-countable.*

▶ **Definition 14** (MaxSAT equivalence)**.** *We say that ϕ_1 and ϕ_2 are MaxSAT-equivalent if, for any truth assignment I on the variables of ϕ_1 and ϕ_2, we have that $cost(I, \phi_1) = cost(I, \phi_2)$.*

▶ **Definition 15.** *A* pseudo-Boolean *(PB) constraint is a Boolean function of the form $\sum_{i=1}^{n} c_i l_i \diamond k$, where k and c_i are integer constants, l_i are literals, and $\diamond \in \{<, \leq, =, \neq, \geq, >\}$.*
A Cardinality (Card) constraint is a PB constraint where all c_i are equal to 1.
An At-Least-One *(ALO) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i \geq 1$.*
An At-Most-One *(AMO) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i \leq 1$.*
An Exactly-One *(EO) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i = 1$.*
An At-Least-K *(ALK) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i \geq k$.*
An At-Mots-K *(AMK) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i \leq k$.*
An Exactly-K *(EK) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i = k$.*

2 Related Work

Paramita is a new framework for SATisfiability-based applications. It shares some similarities with the OptiLog framework [2, 1]. Although OptiLog is already a useful and practical framework, it can also be regarded as a prototype of Paramita. Our experience with the implementation and use of OptiLog helped us identify several areas for improvement. As a result, Paramita has been completely redesigned and reimplemented to provide a significantly more powerful and robust framework.

To place Paramita framework in context, we present an overview of frameworks for solving combinatorial decision and optimization problems. Paramita can also be combined with several of these frameworks to address specific satisfiability-based applications. One possible taxonomy of these frameworks is the following:

General Constraint Programming (CP):

- SCIP [3] (`https://www.scipopt.org/`) is an extensible open-source optimization framework for constraint programming and combinatorial optimization.
- CPMpy (`https://pypi.org/project/cpmpy/`) is a NumPy-based CP modelling framework with direct solver access and backends for OR-Tools, PySAT, Z3, MiniZinc, etc.
- MiniZinc Python [24] (`https://python.minizinc.dev/`) is a high-level constraint modelling language to express and solve discrete optimization problems, with a native Python interface to interact with MiniZinc.
- Numberjack [15] (`https://pypi.org/project/Numberjack/`) is a Python-based modelling language designed to solve complex combinatorial optimization and constraint satisfaction problems (CSPs). It is used to model problems in Python and solve them using backend solvers.
- EasyCP (`https://easycp.sourceforge.net`) is an educational and research-oriented CP framework C++ library.
- PyCSP3 [20] (`https://pycsp.org/`) is a declarative Python framework for modelling combinatorial problems, CSP (Constraint Satisfaction Problem) and COP (Constraint Optimization Problem), and solving them with constraint solvers.
- Savile Row (`https://github.com/savilerow`) is a Python-based constraint modelling system that translates optimization and constraint problems into SAT instances to be solved by SAT solvers.

Mathematical Optimization / CP Hybrid Tools:

- COIN-OR [21] (`https://www.coin-or.org/`) is an open-source collection of optimization libraries and solvers.
- Gurobi [14] (`https://www.gurobi.com/`) and CPLEX [16] (`https://www.ibm.com/products/ilog-cplex-optimization-studio`) are commercial optimization solvers whith APIs for Python, C++, etc.
- OR-Tools [26] (`https://developers.google.com/optimization`)is an open-source optimization framework which include several solvers. The most relevant of them is CP-SAT [25] (`https://developers.google.com/optimization/cp/cp_solver`) which combines Constraint Programming (CP) and Boolean Satisfiability (SAT) solving.
- Pyomo [6] (`https://www.pyomo.org/`) is a Python-based, open-source optimization modelling language. It supports a wide range of problem types, including linear, quadratic, nonlinear programming and mixed-integer linear, quadratic and nonlinear programming.
- PuLP [23] (`https://pypi.org/project/PuLP/`) is a Python library used to create linear programming (LP) and mixed-integer linear programming (MILP) problems. It allows users to define optimization models and use solvers like CBC, GLPK, GUROBI, or CPLEX.
- CVXPY [10] (`https://www.cvxpy.org/`) is an open-source Python-embedded modelling language for convex optimization problems wich lets you express your problem in a natural way.
- DOcplex [17] (`https://pypi.org/project/docplex/`) is a Python library developed by IBM for modelling and solving optimization problems using CPLEX.

SAT / SMT / Logic-Oriented:

- Z3 [9] (`https://github.com/Z3Prover/z3`) is a SMT solver combining and integrating SAT solvers and theory solvers.
- PyPbLib [27] (`https://pypi.org/project/pypblib/`) is a Python library to encode pseudo-Boolean constraints into SAT formulas to be solved by SAT solvers.

- PySAT [18] (`https://pysathq.github.io/`) is a Python library designed for working with Boolean satisfiability (SAT) and MaxSAT problems. It provides a unified interface to several SAT and MaxSAT solvers. It was the first framework, as far as we know, to provide Python bindings for several SAT solvers.
- OptiLog [2, 1] (`https://ulog.udl.cat/static/doc/optilog/html/index.html`) is a Python-based framework for propositional logic and optimization problems. It provides tools to model problems in SAT, MaxSAT, pseudo-Boolean, and related formalisms. It also provide interfaces with external SAT solvers.
- PySMT [12] (`https://pypi.org/project/PySMT/`) is a solver-agnostic library for SMT formulas. It can define several types of formulas such as Linear Real Arithmetic (LRA), Real Difference Logic (RDL), etc. It supports several solvers through native APIs such as MathSAT, Z3, cvc5, Yices 2, CUDD, PicoSAT and Boolector.
- PyNuSMV [5] (`https://pynusmv.readthedocs.io/`) is a Python framework for prototyping and experimenting with BDD-based model checking algorithms based on NuSMV.
- clingo [13] (`https://potassco.org/clingo/`) is an open-source system for Answer Set Programming (ASP), a declarative programming paradigm used to solve complex combinatorial and knowledge-representation problems.
- PyEDA [11] (`https://pyeda.readthedocs.io/`) is a Python library for electronic design automation and Boolean algebra manipulation. It provides tools for working with Boolean expressions, truth tables, logic circuits, and binary decision diagrams (BDDs). It also includes interfaces to SAT solvers.
- XCSP3 [4] (`https://xcsp.org`) is a standard XML-based format for representing Constraint Satisfaction Problems (CSP), Constraint Optimization Problems (COP), and related combinatorial problems.

Combinatorial Planning / Scheduling:

- unified-planning [22] (`https://unified-planning.readthedocs.io`) is a planning framework implemented in Python designed to provide a unified and solver-independent interface for modelling and solving automated planning problems. Once the problem is defined it can be solved with different underlying planners.
- OptaPy [7] (`https://pypi.org/project/optapy`) is a constraint solver in Python that optimizes planning and scheduling problems.

Quantum Computing:

- D-Wave Ocean SDK (`https://docs.dwavequantum.com/en/latest/ocean/`) is a Python-based software development kit for formulating and solving optimization problems using quantum annealing and hybrid quantum-classical methods. It allows users to express problems as QUBO (Quadratic Unconstrained Binary Optimization) or Ising models.
- Qiskit SDK [19] (`https://qiskit-community.github.io/qiskit-optimization/`) is an open-source software development kit for quatum computing. that enables users to create and run quantum algorithms in Python using real quantum computers from IBM or classical simulators.

Finally, Paramita supports several standard C and C++ interfaces for SAT and MaxSAT solvers, namely IPASIR, IPASIR-UP, and IPAMIR:

- IPASIR is the reversed acronym of Re-entrant Incremental SAT Solver API. It is a standard C interface for incremental SAT solvers that allows applications to interact with SAT solvers through a common API by incrementally adding clauses, solving under assumptions, and reusing learned clauses. IPASIR is widely used in formal verification, model checking, and automated reasoning.

- IPASIR-UP extends IPASIR with support for user propagators and advanced interaction between applications and SAT solvers. It enables external control of propagation, conflict explanation, and tighter integration with domain-specific reasoning engines. This extension is useful for SMT solving, constraint programming, and hybrid reasoning systems.
- IPAMIR is a standard incremental interface for MaxSAT solvers. It extends the IPASIR paradigm to optimization problems involving hard and soft clauses. It supports incremental MaxSAT solving, weighted soft constraints, and optimization queries, making it suitable for scheduling, planning, and combinatorial optimization problems.

3 Experimental Evaluation

We conducted an experimental evaluation to analyze the performance of Paramita compared against other state-of-the-art libraries. We used PySAT version 1.9.dev5. All experiments were executed on a computing cluster whose nodes are equipped with two AMD 7402 processors, each comprising 24 cores operating at 2.8 GHz, and 21 GB of RAM per core. For each execution, we imposed a limit of 300 seconds of CPU time and 20 GB of RAM.

3.1 Transforming PB constraints to CNF

In this experiment, we considered all the *DEC-LIN* instances from the PB Competition 2025[2]. For each instance, we load the CNF translation of each PB constraint to a SAT solver. We used the encoder *Best* from PBLib and CaDiCaL version 1.9.5 as SAT solver.

Listings 1 and 2 show the code that we used for PySAT and Paramita respectively.

```
import sys
from collections import defaultdict

from pysat.solvers import Cadical195
from pysat.pb import PBEnc

from pb import read_file, Operator

if __name__ == "__main__":
    max_var = 0

    def new_var():
        global max_var
        max_var += 1
        return max_var

    dimacs_map = defaultdict(new_var)

    s = Cadical195()

    for pb in read_file(sys.argv[1]):
        weights, vars = zip(*pb.terms)
        lits = list(map(lambda v: dimacs_map[v], vars))
        if pb.op == Operator.EQUAL:
            cnf = PBEnc.equals(
                lits=lits, weights=weights,
                bound=pb.bound, top_id=s.nof_vars()
            )
        elif pb.op == Operator.GE:
            cnf = PBEnc.geq(
```

[2] https://www.cril.univ-artois.fr/PB25/

```
                lits=lits, weights=weights,
                bound=pb.bound, top_id=s.nof_vars()
            )
        elif pb.op == Operator.LE:
            cnf = PBEnc.leq(
                lits=lits, weights=weights,
                bound=pb.bound, top_id=s.nof_vars()
            )
        else:
            raise Exception("unreachable")

        s.append_formula(cnf)

    print("Num vars:", s.nof_vars())
    print("Num clauses:", s.nof_clauses())
```

Listing 1 PB loading and conversion script for PySAT.

```
import sys
from collections import defaultdict

from paramita import SatSolver, PbToCnfEncoder, PbConstraint

from pb import read_file, Operator

if __name__ == "__main__":
    max_var = 0

    def new_var():
        global max_var
        max_var += 1
        return max_var

    dimacs_map = defaultdict(new_var)

    s = SatSolver.from_name("Cadical195")
    enc = PbToCnfEncoder.from_name("BestPBLibEncoder")

    for pb in read_file(sys.argv[1]):
        weights, vars = zip(*pb.terms)
        lits = list(map(lambda v: dimacs_map[v], vars))
        op = None
        if pb.op == Operator.EQUAL:
            op = PbConstraint.EQ
        elif pb.op == Operator.GE:
            op = PbConstraint.GTE
        elif pb.op == Operator.LE:
            op = PbConstraint.LTE
        else:
            raise Exception("unreachable")
        enc.encode(PbConstraint(lits, weights, op, pb.bound), s)

    print("Num vars:", s.max_var(), flush=True)
    print("Num clauses:", s.num_clauses(), flush=True)
```

Listing 2 PB loading and conversion script for Paramita.

Figure 1 shows the time consumption of each script with respect to the number of PB constraints.

We found that Paramita converts one more instance than PySAT in the given time limit, and that Paramita is around 50% faster than PySAT, especially in instances with a large amount of PB constraints. In Paramita, all clauses are directly loaded to the SAT solver in C++. In PySAT, the clauses returned by the encoder have to be converted from C++ to Python, and from Python to C++ again to be added to the SAT solver.

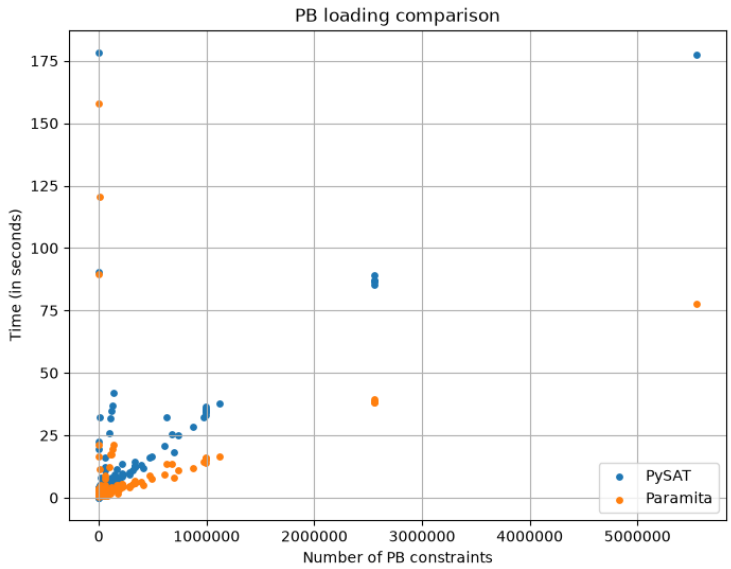


Figure 1 Time consumption results for the PB conversion experiment.

3.2 Cubes generation using `Propagators`

In this section, we generate cubes for SAT instances following the SAT solver heuristic through using `Propagators`, as explained in Section 8 of the article. These cubes could be later used by a cube-and-conquer approach. We selected the instances from the SAT competition 2025 [8].

We executed equivalent codes for PySAT, Paramita in Python and Paramita in C++, using CaDiCaL 1.9.5 as SAT solver. For this last case, we implemented an equivalent version in around 100 lines of C++, as shown in Listing 3.

```cpp
#pragma once

#include <memory>
#include <vector>
#include <string>
#include <unordered_map>
#include <variant>
#include <iostream>

#include <paramita/plugin/ParameterSpace.hpp>
#include <paramita/sat/Propagator.hpp>
#include <paramita/sat/SatSolver.hpp>

using namespace Paramita;

class CubingEngineCpp : public Propagator {
public:
    void on_assignment(const std::vector<int> &lits) override {
        for (const auto &lit : lits) {
            if (solver->is_decision(lit) && std::abs(lit) != blocking_var) {
                trail.push_back(lit);
            }
        }
    }

```

```
26      void on_new_decision_level() override {
27          decision_level++;
28          trlim.push_back(trail.size());
29      }
30
31      void on_backtrack(std::size_t new_level) override {
32          while (trlim.size() > new_level) {
33              while (trail.size() > trlim[trlim.size() - 1]) {
34                  trail.pop_back();
35              }
36              trlim.pop_back();
37          }
38          decision_level = new_level;
39      }
40      bool check_found_model(const std::vector<int> &model) override {
41          return true;
42      }
43
44      bool has_external_clause() override {
45          return trail.size() >= depth;
46      }
47
48      std::vector<int> get_external_clause() override {
49          if (trail.size() >= depth) {
50              m_cubes_.push_back(trail);
51              std::vector<int> clause;
52              clause.push_back(blocking_var);
53              for (const auto &lit : trail) {
54                  clause.push_back(-lit);
55              }
56              return clause;
57          }
58          return {};
59      }
60
61      ParameterSpace get_parameter_space() override {
62          return ParameterSpace(
63              {{"depth", IntegerParameter(3, {1, 1000})},
64               {"blocking_var", IntegerParameter(0, {1, 100000000})},
65               {"solver", SatSolverParameter()}});
66      }
67      void set(const std::string &name, const ParameterType &value) override {
68          if (name == "depth") {
69              depth = std::get<int>(value);
70          }
71          else if (name == "blocking_var") {
72              blocking_var = std::get<int>(value);
73          }
74          else if (name == "solver") {
75              solver = std::get<std::shared_ptr<SatSolver>>(value);
76          }
77          else {
78              throw std::runtime_error("Unknown parameter.");
79          }
80      }
81
82      std::vector<std::string> get_out_data_keys() const override {
83          return {"cubes_"};
84      };
85      OutData::Value get_out_data(const std::string &key) const override {
86          if (key == "cubes_") {
87              return OutData::to_value(m_cubes_);
88          }
89          return nullptr;
90      }
91
```

```
private:
    int depth = 3;
    int blocking_var = 0;
    std::shared_ptr<SatSolver> solver;

    std::vector<std::vector<int>> m_cubes_;

    int decision_level = 0;
    std::vector<int> trail;
    std::vector<int> trlim;
};
```

Listing 3 Paramita C++ implementation of the Cubes Propagator.

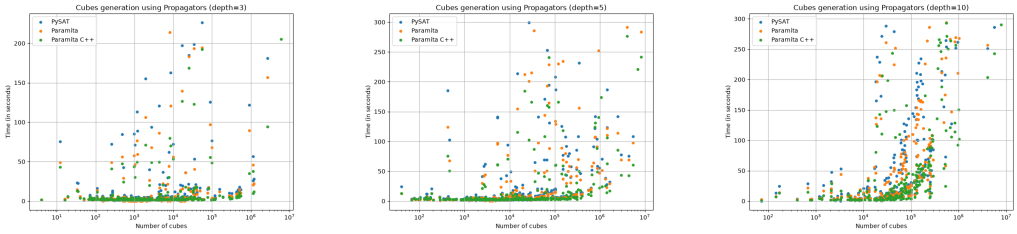
This C++ propagator can be used in Python through the code in Listing 4:

```
import sys

from paramita import SatSolver, Propagator, load_cnf


def main():
    s = SatSolver.from_name("Cadical195")
    load_cnf(sys.argv[1], s)
    print("Instance loaded")
    depth = int(sys.argv[2])

    print("Num vars:", s.max_var(), flush=True)
    print("Num clauses:", s.num_clauses(), flush=True)

    blocking_var = s.max_var() + 1

    prop = Propagator.dynamic_load("build/libcubing-engine-cpp.so")
    prop.set("depth", depth)
    prop.set("blocking_var", blocking_var)
    prop.set("solver", s)
    s.connect_external_propagator(prop)
    vars = [i + 1 for i in range(s.max_var())]
    for v in vars:
        s.add_observed_var(v)
    s.add_observed_var(blocking_var)

    is_sat = s.solve([-blocking_var])
    print("Is sat:", is_sat, flush=True)
    print(f"Num cubes: {len(prop.cubes_)}", flush=True)


if __name__ == "__main__":
    main()
```

Listing 4 Usage of the C++ Propagator implementation in Python.

Figures 2a, 2b and 2c show the time consumption results for all the methods using depths 3, 5, and 10 respectively, with respect to the number of generated cubes (which we ensured is the same for all the three approaches).

In general, Paramita Python version is slightly faster than PySAT, being able to finish 1 more instance for depth 3, 9 more instances for depth 5 and 6 more instances for depth 10. The greatest difference is when comparing Paramita C++ version, which compared to PySAT finishes 2 more instances for depth 3, 11 more instance for depth 5 and 29 more instances for depth 10, always obtaining better running times in all the instances solved by both methods.



(a) Depth 3. **(b)** Depth 5. **(c)** Depth 10.

Figure 2 Time consumption results for cubes generation experiments using depths 3, 5, and 10.

References

1 Josep Alòs, 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)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2022.25`, `doi:10.4230/LIPIcs.SAT.2022.25`.

2 Carlos Ansótegui, Jesús Ojeda, Antonio Pacheco, Josep Pon, Josep M. Salvia, and Eduard Torres. Optilog: A framework for sat-based systems. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 1–10, Cham, 2021. Springer International Publishing.

3 Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Liding Xu. The scip optimization suite 9.0, 2024. URL: `https://arxiv.org/abs/2402.17702`, `arXiv:2402.17702`.

4 Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: An integrated format for benchmarking combinatorial constrained problems, 2024. URL: `https://arxiv.org/abs/1611.03398`, `arXiv:1611.03398`.

5 Simon Busard and Charles Pecheur. Pynusmv: Nusmv as a python library. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, pages 453–458, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-38088-4_33`.

6 Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, third edition, 2021. `doi:10.1007/978-3-030-68928-5`.

7 Christopher Chianelli and Geoffrey De Smet. Optapy: An ai constraint solver for python. URL: `https://github.com/optapy/optapy`.

8 Cayden Codel, Katalin Fazekas, Marijn J. H. Heule, Markus Iser, Cayden Codel, Katalin Fazekas, Marijn J. H. Heule, and Markus Iser. Proceedings of SAT Competition 2025 : Solver and Benchmark Descriptions. Technical report, TU Wien, August 2025. URL: `https://repositum.tuwien.at/handle/20.500.12708/218424`, `doi:10.34726/10379`.

9 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

10 Steven Diamond and Stephen Boyd. Cvxpy: a python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.*, 17(1):2909–2913, January 2016.

11 Chris Drake. PyEDA: Python Electronic Design Automation, 2015. URL: https://github.com/cjdrake/pyeda.

12 Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 1–12, 2015.

13 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = asp + control: Preliminary report, 2014. URL: https://arxiv.org/abs/1405.3694, arXiv:1405.3694.

14 Gurobi Optimization, LLC, 2026. URL: https://www.gurobi.com.

15 Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan. Constraint programming and combinatorial optimisation in numberjack. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 181–185, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

16 IBM. IBM ILOG CPLEX Optimization Studio, 2026. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio.

17 IBM. IBM® decision optimization CPLEX® modeling for python, 2026. URL: https://ibmdecisionoptimization.github.io/docplex-doc/.

18 Alexey Ignatiev, Antonio Morgado, and Joao 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*, pages 428–437, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94144-8_26.

19 Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with qiskit, 2024. URL: https://arxiv.org/abs/2405.08810, arXiv:2405.08810.

20 Christophe Lecoutre and Nicolas Szczepanski. Pycsp3: Modeling combinatorial constrained problems in python, 2024. URL: https://arxiv.org/abs/2009.00326, arXiv:2009.00326.

21 Robin Lougee. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47:57 – 66, 02 2003. doi:10.1147/rd.471.0057.

22 Andrea Micheli, Arthur Bit-Monnot, Gabriele Röger, Enrico Scala, Alessandro Valentini, Luca Framba, Alberto Rovetta, Alessandro Trapasso, Luigi Bonassi, Alfonso Emilio Gerevini, Luca Iocchi, Felix Ingrand, Uwe Köckemann, Fabio Patrizi, Alessandro Saetti, Ivan Serina, and Sebastian Stock. Unified planning: Modeling, manipulating and solving ai planning problems in python. *SoftwareX*, 29:102012, 2025. URL: https://www.sciencedirect.com/science/article/pii/S2352711024003820, doi:10.1016/j.softx.2024.102012.

23 Stuart Mitchell, Michael O'Sullivan, and Iain Dunning. Pulp: A linear programming toolkit for python, 2011. URL: https://optimization-online.org/2011/09/3178/.

24 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74970-7_38.

25 Laurent Perron and Frédéric Didier. CP-SAT. URL: https://developers.google.com/optimization/cp/cp_solver/.

26 Laurent Perron and Vincent Furnon. OR-Tools. URL: https://developers.google.com/optimization/.

27 Tobias Philipp and Peter Steinke. Pblib – a library for encoding pseudo-boolean constraints into cnf. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 9–16, Cham, 2015. Springer International Publishing.