# `MiniSAT(ID)` for satisfiability checking and constraint solving

Broes De Cat and Bart Bogaerts and Marc Denecker
Knowledge Representation and Reasoning
KU Leuven
marc.denecker@cs.kuleuven.be

September 25, 2014

`MiniSAT(ID)` is an engine for satisfiability checking and finite domain constraint solving. It solves problems expressed in the quantification-free language ECNF, an extension of CNF. The system holds a middle ground between the emerging field of Constraint Answer Set Programming (CASP) [10] and Constraint Programming (CP) [11]. The language ECNF is a ground fragment of the language `FODOT`. The latter is an extension of first order logic (FO) and includes types, inductive definitions, aggregates, uninterpreted functions and (bounded) arithmetic. `FODOT` is related to the family of Answer Set Programming languages. `MiniSAT(ID)` implements model generation inference, taking as input an ECNF theory $T_g$ and returns models of it, assignments to its symbols that satisfy $T_g$ according to `FODOT`'s formal semantics.

`MiniSAT(ID)` is a kernel component of the knowledge base system `IDP`. The latter provides multiple forms of inference and a `Lua`-based programming environment for `FODOT`. A key inference of `IDP` is model expansion, a generalization of Herbrand model generation. It takes as input a theory $T$ and a partial structure $\mathcal{I}$ and returns models $M$ of $T$ expanding $\mathcal{I}$, or UNSAT if no such models exists. An extension is optimization inference, which has a numerical term as extra argument and returns models with a minimal value for this term. Model expansion in `IDP` operates by grounding $T$ in $\mathcal{I}$ to a ground ECNF theory $T_g$ and running `MiniSAT(ID)` on $T_g$. This is a similar *ground and solve* strategy as found in ASP systems as well as in solvers of expressive constraint languages such as `Zinc` [11].

ECNF integrates aspects from ground languages of SAT, ASP and CP. An ECNF theory consists of ground clauses $L_1 \vee \ldots \vee L_n$ and definitional rules $A \leftarrow B$ with head $A$ a ground atom and body $B$ either a conjunction or disjunction of literals, or a complex atom. Complex atoms are either aggregate expressions (sum, cardinality, min, max, product) or constraints on uninterpreted constants. In CP terminology such constants are "variables". They may appear in the head of definitional rules and in complex atoms and have an associated domain. The use of such variables can reduce grounding size significantly. ECNF shares aspects from `ASPcore-2` as well as `FlatZinc` [13].

The development of `MiniSAT(ID)` is part of a larger trend, also apparent in the emerging field of Constraint ASP, to improve search by combining ideas from different fields such as SAT, CP and ASP. The system was built on top of the famous `MiniSAT` solver and natively combines CDCL search with efficient propagation for uninterpreted functions, arithmetic, aggregates and inductive definitions. All non-propositional symbols and expressions in ECNF are "hidden" within the definitional part of the theory, so that standard SAT solving algorithms can operate on the clausal part of the theory. The SAT solving process is interleaved with calls to the propagators of the special language constructs. All propagators in `MiniSAT(ID)` are based on the technique of *Lazy Clause Generation* [14]. This technique creates, for each propagation performed by a propagator on a variable, a CNF clause that "explains" this propagation, and adds it to the clausal theory. This clause can later be used for conflict-driven clause learning, intelligent back-jumping and propagation. It combines the simplicity and power of the SAT CDCL technology with CP technology. An essential feature of `MiniSAT(ID)` is that new symbols and rules can be added dynamically during search. This aspect is vital to enable Lazy Clause Generation and for the related technique of Lazy Grounding.

**Example 1** *Consider the following birthday riddle : "To determine my age, it suffices to know that my current age in 2013 is halfway between two consecutive primes, that my age's prime factors do not sum to a prime number, and that I was born in a prime year.". In* `FODOT`*, it can be modeled as:*

```
vocabulary V is {
    type Nb isa int;
    func Age[->Nb];
    pred Prime(Nb);
    func YearOfBirth[->Nb];
}
theory T over V is {
    { Prime(x) <- x>1 & !y: 1 < y < x => ~ (x % y = 0) }

    Age = 2013-YearOfBirth;
    Prime(YearOfBirth);

    ?x1 x2: Prime(x1) & Prime(x2) & x1 < Age < x2 &
            ~(?y: Prime(y) & x1 < y < x2) & Age = (x2 + x1)/2;

    ~Prime(sum { x : Prime(x) & 1 < x =< Age & Age % x = 0 : x });
}
structure S over V is { Nb = {0..2013} }
```

`IDP` *is unable to ground this theory to ECNF without uninterpreted constants due to memory exhaustion. With uninterpreted constants,* `IDP` *takes half a second*

| Benchmark | # solved IDP | # solved Gringo-Clasp |
|---|---|---|
| Perm. P. Matching | 10 | 10 |
| Valves Location * | **7** | 4 |
| Still-Life * | 2 | **3** |
| Graceful Graphs | 3 | **9** |
| Bottle Filling | 10 | 10 |
| NoMystery | **9** | 6 |
| Sokoban | **7** | 5 |
| Ricochet Robots | 7 | **10** |
| Crossing Minim. * | 0 | **9** |
| Solitaire | 8 | **9** |
| Weighted Sequence | 10 | 10 |
| Stable Marr. | 10 | 10 |
| Incremental Sched. | **6** | 5 |
| Visit All $_{core}$ | 6 | **7** |
| Knight's Tour $_{core}$ | **1** | 0 |
| Maximal Clique $^{*}_{core}$ | 0 | **1** |
| Graph Col. $_{core}$ | **7** | 4 |

Table 1: Experimental results for benchmarks of the 2013 ASP competition. For optimization problems (*), # solved is the number of instances for which optimality was proven. Winners are shown in bold.

*to find a solution. In fact,* IDP *proves that 48 different solutions exist; however only one is an age below 100, namely* $Age = 26$.

**Experiment with IDP as an ASP System.** In 2013, IDP (grounder and MiniSAT(ID)) participated in the ASP competition [1] in the Model-and-Solve Track and ran fourth on seven participants. Because it had been disqualified on several benchmarks due to modeling errors, we reran the competition benchmarks with IDP and the winner Gringo-Clasp of the Potsdam ASP group. The results are displayed in Table 1. The table contains also four benchmarks of the System Track (annotated by $_{core}$).

The results show that Gringo-Clasp solved more instances than IDP (122 instances against 113) and often required less time to solve an instance (not shown). IDP solved more instances in 6 out of 17 benchmarks. Recall that in the Model and Solve Track, IDP and Gringo-Clasp were run on different encodings. The encodings for IDP tend to be simpler, less fine-tuned than those of Gringo-Clasp. For instance, for Connected, Maximum Density Still-Life, 50 lines of FODOT against 100 for Gringo-Clasp; for Crossing Minimization, 10 lines of FODOT against 50, including a sophisticated symmetry breaking axiom that performed very well. This certainly is part of the explanation why IDP was outperformed on some of these benchmarks. In the *core* benchmarks where both systems solved similar encodings there are no large discrepancies between both systems.

| Solver | AST (sec.) | PSI (%) |
|---|---|---|
| minisatid | 950.91 | 51.62 |
| g12cpx | 1126.98 | 41.68 |
| fzn2smt | 1143.47 | 38.13 |
| ortools | 1316.25 | 30.65 |
| g12lazyfd | 1306.10 | 30.31 |
| gecode | 1354.65 | 29.51 |
| izplus | 1350.42 | 28.05 |
| bprolog | 1423.45 | 24.73 |
| jacop | 1435.123 | 24.67 |
| g12fd | 1424.80 | 23.57 |
| mistral | 1525.83 | 16.91 |
| g12mip | 1597.54 | 12.58 |

Table 2: Experimental evaluation of MiniZinc solvers on the CSPs in Benchmark Set B [2].

Although we cannot easily draw conclusions from this table, the results suggest that `IDP` performs quite well in comparison to other ASP systems. Specifically for more natural encodings, the various analysis tools and automatic transformations in `IDP` turn out to be an important advantage. It is part of future work to implement an `ASPcore-2` parser; this will enable us to run `IDP` on the same encodings as ASP solvers and allow a more objective comparison.

**Experiment with `MiniSAT(ID)` as a `MiniZinc` Solver.** In the context of developing a `MiniZinc` portfolio system, Amadini et al. [2] compared 12 different `MiniZinc` solvers on a data set of 4642 Constraint Satisfaction Problems. In the case of `MiniSAT(ID)` and several other solvers, the tool `mzn2fzn` was run as a preprocessor to reduce `MiniZinc` specifications to `FlatZinc`. The results are shown in Table 2.[1] For each solver, the table presents the Average Solving Time (AST) and the Percentage of Solved Instances (PSI). `MiniZinc` specifications can contain heuristic information and global constraints that solvers can exploit to improve search; however, this information is ignored by `MiniSAT(ID)`, which always applies its domain-independent heuristic and a standard translation of global constraints. The table allows us to conclude that `MiniSAT(ID)` is the best performing `MiniZinc`-system of those compared, with a smaller average solving time than any other system and solving 10% more benchmarks than the runner-up (g12cpx). [2]

[1] Courtesy of Roberto Amadini and colleagues.

[2] In a more recent experiment, a new version of `MiniSAT(ID)` participated in the `MiniZinc` challenge [12], testing systems both on constraint satisfaction and constraint optimization problems. The new system performed poorly. Analysis revealed that a major culprit was the extensive use of an arbitrary precision integer module, which slowed down the system significantly.

**Conclusion and further information.** MiniSAT(ID) incorporates state-of-the-art technology from SAT, ASP and CP. It is designed to be an extensible search framework that allows developers to easily extend the input language and plug in new propagators. Other current features of the solver are dynamic symmetry breaking [7] and an interface to tightly integrate it with a grounder to allow for *Lazy Grounding*. The latter boils down to interleaving grounding and solving so that solutions can be found without fully grounding a theory [6]. MiniSAT(ID) supports a variety of input and output languages. The implementation is currently one of the best free-search MiniZinc solvers and is on-par (although less rich in features) with the award-winning ASP solver Clasp. It is also one of the first open-source implementations of Lazy Clause Generation.

IDP and MiniSAT(ID) can be downloaded from [8]. Information about FODOT and the IDP system is available at [4]. A technical description of MiniSAT(ID) and its main contributions has been published in [5] and elaborated upon in [3]. A webpage is available to interactively run IDP at [9].

# References


[1] Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, Jörg Pührer, Christoph Redl, Francesco Ricca, Patrik Schneider, Martin Schwengerer, Lara Katharina Spendier, Johannes Peter Wallner, and Guohui Xiao. The fourth Answer Set Programming competition: Preliminary report. In Pedro Cabalar and Tran Cao Son, editors, *LPNMR*, volume 8148 of *LNCS*, pages 42–53. Springer, 2013.

[2] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving csps. In Carla P. Gomes and Meinolf Sellmann, editors, *CPAIOR*, volume 7874 of *Lecture Notes in Computer Science*, pages 316–324. Springer, 2013.

[3] Broes De Cat. Separating Knowledge from Computation: An FO(.) Knowledge Base System and its Model Expansion Inference. Phd thesis, KU Leuven, Belgium, 2014.

[4] Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312, 2014.

[5] Broes De Cat, Bart Bogaerts, Jo Devriendt, and Marc Denecker. Model expansion in the presence of function symbols using constraint programming. In *ICTAI*, pages 1068–1075. IEEE, 2013.

[6] Broes De Cat, Marc Denecker, Peter J. Stuckey, and Maurice Bruynooghe. Lazy model expansion: Interleaving grounding with search. *CoRR*, abs/1402.6889, 2014.

[7] Jo Devriendt, Bart Bogaerts, Broes de Cat, Marc Denecker, and Christopher Mears. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *ICTAI*, pages 49–56. IEEE, 2012.

[8] The IDP system. `http://dtai.cs.kuleuven.be/krr/software`, 2013.

[9] The IDP web-IDE. `http://http://seldon.cs.kuleuven.be/idp`, 2014.

[10] Yuliya Lierler. On the relation of constraint answer set programming languages and algorithms. In Jörg Hoffmann and Bart Selman, editors, *AAAI*. AAAI Press, 2012.

[11] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[12] Minizinc challenge 2014. `http://www.minizinc.org/challenge2014/results2014.html`.

[13] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *CP'07*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[14] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.