The return of *xorro*

Flavio Everardo[1][0000−0002−6421−3158]

University of Potsdam, Germany
flavio.everardo@cs.uni-potsdam.de

Parity constraints constitute the foundations of reasoning modes like sampling or (approximate) model counting [5], as well as circuit verification and cryptography [4]. With most of their applications in the neighboring area of Satisfiability Testing (SAT) [5], almost no attention has so far been paid to their integration into Answer Set Programming (ASP).

Previous efforts represented parity constraints into ASP in three ways, via the `#count` aggregate coupled with a modulo-two operation as used or sampling in the initial prototype of *xorro* from 2009[1], as lists, as shown in *harvey* [3], and as the (discontinued) aggregates `#even` and `#odd` from *gringo* series 3 via meta-encodings. Unlike these approaches, several SAT solvers feature rather sophisticated treatments of parity constraints. For instance, most popularly the award-winning solver *crypto-minisat* [6], which pursues a hybrid approach, addressing parity constraints separately with Gauss-Jordan Elimination (GJE).

To this end, we present the next generation of *xorro*, [7] implementing six alternatives to handle parity constraints into ASP, benefiting from the advanced interfaces of *clingo*, and the sophisticated solving techniques developed in SAT. We propose two types of approaches, eager and lazy. [2] The former relies on ASP encodings of parity constraints, and the latter uses theory propagators within *clingo*'s Python interface [2].

To accommodate parity constraints in the input language, we rely on *clingo*'s theory language extension [2] following the common syntax of *aggregates* [1]:

```
1   &odd{ 1 : p(1) }.
2  &even{ X : p(X), X>1 }.
```

That is, *xorro* extends the input language of *clingo* by aggregate names `&even` and `&odd` that are followed by a set, whose elements are *terms* conditioned by conjunctions of literals separated by commas.[3] In the context of a choice rule `{p(1..3)}.`, the parity constraints shown above amounts to the XOR operations: $p(1) \oplus \bot$ and $p(2) \oplus p(3) \oplus \top$ (where $\bot$ and $\top$ stand for the Boolean constants true and false, respectively) yielding the answer sets {`p(1)`} and {`p(1),p(2),p(3)`}.

Currently, these constraints are interpreted as directives, filtering answer sets that do not satisfy the parity constraint in question. [4] Hence, the first constraint filters out answer sets not containing the atom `p(1)`, while the second requires that either none or both of the atoms `p(2)` and `p(3)` are included.

[1] https://sourceforge.net/p/potassco/code/HEAD/tree/branches/xorro
[2] Both eager and lazy follows the methodology from Satisfiability modulo theories.
[3] In turn, multiple conditional terms within an aggregate are separated by semicolons.
[4] For now, parity constraints may not occur in the bodies nor the heads of rules.

Table 1, shows the six implementations to handle parity constraints. The first three corresponds to the eager, and the last three to the lazy approaches.

Approach	Description
count	Add count aggregates with a modulo 2 operation
list,tree	Translate binary XOR operators into rules forming list and tree structures
countp	Propagator simply counting truth literals on total assignments
up	Propagator implementing unit propagation
gje	Propagator implementing (non-incremental) Gauss-Jordan Elimination

Table 1. *xorro* approaches to handle parity constraints

Finally, we empirically evaluate the different approaches in view of their impact on solving performance, while varying the number and size of parity constraints compared against *clingo* solving time. The experiments show that *xorro* scales depending on the combination of the number, density, and preprocessing of the parity constraints. When increasing the number of high-density constraints as used in sampling (XORs with a size of half the program variables), we start to see that the solving time increases concerning *clingo*. Comparing to previous approaches, the eager counting, and the list (from the previous *xorro* and *harvey*), both stay behind for sampling purposes. Their scalability is subjected to the density of the parity constraints and preprocessing, and particularly, grounding becomes the bottleneck for the counting approach with aggregates.

References

1. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**(4-5), 449–463 (2015)
2. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Carro, M., King, A. (eds.) Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16). vol. 52, pp. 2:1–2:15. Open Access Series in Informatics
3. Greßler, A., Oetsch, J., Tompits, H.: Harvey: A system for random testing in ASP. In: Balduccini, M., Janhunen, T. (eds.) Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17). Lecture Notes in Artificial Intelligence, vol. 10377, pp. 229–235. Springer-Verlag.
4. Laitinen, T.: Extending SAT Solver with Parity Reasoning. Dissertation, Aalto University (Nov 2014)
5. Meel, K.: Constrained Counting and Sampling: Bridging the Gap between Theory and Practice. Dissertation, Rice University (Aug 2018)
6. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09). Lecture Notes in Computer Science, vol. 5584, pp. 244–257. Springer-Verlag (2009)
7. Everardo, F., Janhunen, T., Kaminski, R., Schaub, T.: The return of *xorro*. In: Balduccini M., Lierler Y., and Woltran S. (eds.) Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19). Lecture Notes in Artificial Intelligence, vol. 11481, pp. 284–297. Springer-Verlag (2019)