Parity constraints constitute the foundations of reasoning modes like sampling or (approximate) model counting \cite{5}, as well as circuit verification and cryptography \cite{4}. With most of their applications in the neighboring area of Satisfiability Testing (SAT) \cite{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 \texttt{#count} aggregate coupled with a modulo-two operation as used or sampling in the initial prototype of xorro from 2009\footnote{https://sourceforge.net/p/potassco/code/HEAD/tree/branches/xorro}, as lists, as shown in harvey \cite{3}, and as the (discontinued) aggregates \texttt{#even} and \texttt{#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 \cite{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, \cite{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.\footnote{Both eager and lazy follows the methodology from Satisfiability modulo theories.} The former relies on ASP encodings of parity constraints, and the latter uses theory propagators within clingo’s Python interface \cite{2}.

To accommodate parity constraints in the input language, we rely on clingo’s theory language extension \cite{2} following the common syntax of aggregates \cite{1}:

\begin{verbatim}
1   &odd{ 1 : p(1) }.
2   &even{ X : p(X), X>1 }.
\end{verbatim}

That is, xorro extends the input language of clingo by aggregate names \texttt{&even} and \texttt{&odd} that are followed by a set, whose elements are \textit{terms} conditioned by conjunctions of literals separated by commas\footnote{In turn, multiple conditional terms within an aggregate are separated by semicolons.}. 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.\footnote{For now, parity constraints may not occur in the bodies nor the heads of rules.} 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.
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.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Add count aggregates with a modulo 2 operation</td>
</tr>
<tr>
<td>list, tree</td>
<td>Translate binary xor operators into rules forming list and tree structures</td>
</tr>
<tr>
<td>countp</td>
<td>Propagator simply counting truth literals on total assignments</td>
</tr>
<tr>
<td>up</td>
<td>Propagator implementing unit propagation</td>
</tr>
<tr>
<td>gje</td>
<td>Propagator implementing (non-incremental) Gauss-Jordan Elimination</td>
</tr>
</tbody>
</table>

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