Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 354 (2020) 29–44

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Towards an Answer Set Programming Methodology for Constructing Programs Following a Semi-Automatic Approach – Extended and Revised version

Flavio Everardo[a,[0000−0002−6421−3158],1] Mauricio Osorio[b]

[a] *University of Potsdam, Germany*
Email: flavio.everardo@cs.uni-potsdam.de

[b] *Universidad de las Americas Puebla, Mexico*
Email: osoriomauri@gmail.com

Abstract

Answer Set Programming (ASP) is a successful rule-based formalism for modeling and solving knowledge-intense combinatorial (optimization) problems. Despite its success in both academic and industry, open challenges like automatic source code optimization, and software engineering remains. This is because a problem encoded into an ASP might not have the desired solving performance compared to an equivalent representation. Motivated by these two challenges, this paper has three main contributions. First, we propose a developing process towards a methodology to implement ASP programs, being faithful to existing methods. Second, we present ASP encodings that serve as the basis from the developing process. Third, we demonstrate the use of ASP to reverse the standard solving process. That is, knowing answer sets in advance, and desired strong equivalent properties, "we" exhaustively reconstruct ASP programs if they exist. This paper was originally motivated by the search of propositional formulas (if they exist) that represent the semantics of a new aggregate operator. Particularly, a parity aggregate. This aggregate comes as an improvement from the already existing parity (XOR) constraints from *xorro*, where lacks expressiveness, even though these constraints fit perfectly for reasoning modes like sampling or model counting. To this end, this extended version covers the fundaments from parity constraints as well as the *xorro* system. Hence, we delve a little more in the examples and the proposed methodology over parity constraints. Finally, we discuss our results by showing the only representation available, that satisfies different properties from the classical logic XOR operator, which is also consistent with the semantics of parity constraints from *xorro*.

Keywords: nswer set programming, Combinatorial optimization problems, parity aggregate operator

1 Introduction

Answer Set Programming (ASP; [5]) is a rule-based formalism for modeling and solving knowledge-intense combinatorial (optimization) problems. ASP's attractiveness consists of the combination of a declarative modeling language with highly

[1] Affiliated with Tecnológico de Monterrey campus Puebla, Mexico.

https://doi.org/10.1016/j.entcs.2020.10.004

effective solving engines, allowing to specifying a given (search) problem rather than programming the algorithm for solving it. In other words, given a search problem, a programmer specifies the search space domain and problem-specific properties. Combined, let an ASP solver propose solutions called answer sets.

Currently, ASP is robust and mature enough, offering many important language constructs like aggregation, (weak) constraints, different types of negations, and optimization statements to mention a few, as well as high-performance solvers. An example of a *state-of-the-art* and *award-winning* ASP solvers is *clasp* [7] demonstrating its competitiveness and versatility, by winning first places at various solver contests since 2011 (eg. ASP, CASC, MISC, PB, and SAT competitions). [2] *clasp*, combined with the grounder *gringo* [9], composes *clingo* [10], an ASP system to ground and solve logic programs. For this article, we expect the reader to be familiar with ASP. For technical aspects, including theoretical works, implementations, and applications, see [11,8,10]. On the other hand, we refer to [13] as an introductory yet complete reading of ASP.

Despite the success of ASP in both academic and industry, [3] in areas like planning, scheduling, configuration, design, and diagnosis (to mention a few), challenges like automatic source code optimization, and software engineering remain open, where there is a need to integrate software engineering methodologies and tools into ASP [4], where, fortunately, they come hand in hand.

Motivated by these two challenges, this paper has three main contributions. First, we propose a developing process towards a methodology to implement ASP programs, being (as much as possible) faithful to the method proposed by [14]. Second, we present ASP encodings that fall under the category of *meta-programs* [12] serving as the basis from our developing process. Third, we demonstrate the use of ASP to reverse the standard solving process. That is, knowing answer sets in advance, and desired strong equivalent properties, exhaustively reconstruct ASP programs if they exist, following the approaches from [24,25].

It is relevant to remark that this paper is an extended version from [6], where the proposed methodology towards a more generalized framework, is a consequence of the search of different propositional formulas (if they exist) that represent the semantics of a new aggregate operator. Especially, this operator seeks to behave as a parity aggregate, as an improvement from the already existing parity constraints from *xorro*. Even though these constraints fit perfectly for reasoning modes like sampling or model counting [42,41,39,40,43], they lack expressiveness for other applications also in the neighboring area of Satisfiability Testing (SAT) [2] like cryptography [45,3].

To this end, the remainder of the paper is structured as follows. In Section 2, we focus on the introduction of non-standard concepts used in [6], for the ASP program implementations like, the formal definition of *Strong Equivalence* (SE), the approach to construct formulas from an interpretation in the 3-valued logic of G_3, which is straightforward related with the logic of Here-and-There (HT). Also,

[2] For more details of clasps trophies and tracks, see http://potassco.sourceforge.net/trophy.html.

[3] An incomplete but vast list of ASP applications:
https://www.dropbox.com/s/pe261e4qi6bcyyh/aspAppTable.pdf

we cover the best practices for designing and developing ASP programs from a software engineering perspective. We close this section with a brief description of the formalities of parity constraints as well as the *xorro* system. To set the context of the intended process, Section 3 illustrates a running example using ASP as an overview to reverse the standard solving process, followed by a more fine-grained details including ASP codifications around parity constraints. From there, we discuss our results by showing the only representation available, that satisfies different properties from the classical logic XOR operator, which is also consistent with the semantics of parity constraints from *xorro*. Lastly, we conclude the paper and direct future work in Section 4.

2 Background

In this section, we present theoretical and practical aspects that would be of interests in our proposed approach such as formal definition of strong equivalence, the formalities to construct propositional formulas using Gödel's 3-valued logic (G_3) [28], and its straightforward relationship with the logic of Here-and-There (HT) [27]. We recapitulate the design and development process of ASP programs (which has been proven in the industry) from [14]. Lastly, we finish this section with the fundaments of XOR constraints as well as a brief description of the *xorro* system.

2.1 Strong Equivalence

The term *Strong Equivalence* [21], concerning ASP programs, means that, having two programs (formulas) F_1 and F_2, F_1 is strongly equivalent (SE) to F_2 if F_1 is equivalent to F_2 in the Gödel's 3-valued logic (G_3), which is commensurate to the logic of *Here-and-There* (HT). Also, via the *reduct* [31], F_1 is SE to F_2 if for each set X of atoms both *reducts* F_1^X and F_2^X are equivalent in classical logic [22,23]. It is relevant to remark the importance of *Strong Equivalence* into a software engineering perspective, which not only F_1 and F_2 comprise the same answer sets (meaning $F_1 \equiv F_2$) but, we can extend both formulas with another one R such that $F_1 \cup R$ and $F_2 \cup R$ yield the same answer sets (represented by $F_1 \equiv_{SE} F_2$).

2.2 Constructing formulas from an interpretation in G_3 or HT

For software engineering purposes, it is possible to construct propositional formulas (hence, ASP programs) from an interpretation in HT [24,25]. Yet, it is also possible to use G_3 logic, which it is equivalent to HT, and the relationship is straightforward. [4]

For the G_3 values 0, 1, and 2, 0 equals $\bot$ or false *There*, 1 equals false *Here* but true *There*, and 2 equals $\top$ or true *Here*. Therefore, considering that both logics G_3 and HT are equivalent, we keep G_3 for the remainder of the paper. That is, given a

[4] We only mention the needed concepts from the logics of G_3 and HT. For more information, we may refer the reader to [32,33].

G_3-interpretation I, we apply the following specification or clause C from [25]. [5] To create a clause, we apply the formula below whenever an interpretation equals to 2. A more detailed example is shown in Table 3 from Section 3.

$$\left(\bigwedge_{I(v)=2} v\right) \wedge \left(\bigwedge_{I(w)=0} \neg w\right) \wedge \left(\bigwedge_{I(x)=1} \neg\neg x\right) \wedge \left(\bigwedge_{I(y),I(z)=1,y\neq z} (y \rightarrow z)\right) \quad (1)$$

Then, to construct the propositional formula, we need to apply disjunctions over the resulting clauses. This propositional formula can be simplified according to (but not necessarily all) [31,34,35]. For a concrete example we refer to Section 3.1.

On the other hand, we can apply the same procedure to find a counter-example for two programs P_1 and P_2 such that $P_1 \equiv P_2$ (yield the same answer sets), but instead of applying disjunctions over the resultant clauses, we conjunct them [30]. This counter-example serves to prove if both programs are strong equivalent, meaning that $P_1 \equiv_{SE} P_2$ as shown in the previous section.

2.3 Software Engineering

The work from [14] proposes a six steps methodology for the development of ASP programs, following the project management (PM) standard ISO 21500:2012, also coordinated with the principles behind the life cycles development from the Project Management Body of Knowledge (PMBOK) [26]. Next, we summarize the six areas and let us point out the intersection with the stages from our methodology in Section 3.

(i) **Identify the needs** Find opportunities where ASP is stronger than conventional methods. Define and document the application requirements properly (first stage).

(ii) **Design a valid specification of the problem** Implement an ASP specification with small instances for testing (second stage with support from the first stage).

(iii) **Performance engineering** Explore alternatives of ASP program implementations (third stage with the support of the first two stages), and evaluate their performance considering "real-world" size instances. [6] For a prototyping process, like our methodology, we focus more on readability rather than performance.

(iv) **Integrate into the existing environment** Choose the best ASP program alternative from the feasibility study.Evaluate other solving strategies, and answer sets handling. Design interfaces towards a complete implementation, benefiting standard libraries and API from ASP solvers like *clingo* in languages like Python or C++.

[5] The original formula is in the context of HT. For consistency, we adapt it to G_3.

[6] Particularly, for the third area, we only focus on the exploration of ASP program alternatives and their implementations. Their performance evaluation concerning "real-world" size instances, is left for future work.

(v) **Testing and debugging** Ensure high-quality via automated tests, and debugging of ASP programs if applicable. For instance, ASP Debbuggers like [18,19].

(vi) **Maintenance** Focus on a well-defined structure of the program, and benefit from ASP's modularity for further adaptions.

Also, [14] stated that in this development process, they consider knowledge base design and performance engineering as the most important and most different steps from conventional software engineering. Our method falls in these two steps, particularly, covering the first three, letting glimpse opportunities to develop the last three steps.

Furthermore, in [14], they use an Object-Oriented approach (OO) into ASP called OOASP, which allows analyzing OO software models and their instances employing ASP. The OOASP approach has been successfully implemented in Siemens, as an extension to any OO modeling environment. It has been evaluated together with Siemens internal tools. This modeling approach is currently out of the scope of this paper, but it will be considered for future work development.

2.4 Parity (XOR) Constraints

Towards the definition of parity constraints, let $\top$ and $\bot$ stand for the Boolean constants *true* and *false*, respectively. Given atoms a_1 and a_2, the *exclusive or* (XOR for short) of a_1 and a_2 is denoted by $a_1 \oplus a_2$ and it is satisfied if *either* a_1 *or* a_2 is true (but not both). Due to associativity, we can generalize the idea of n distinct atoms $a_1, \ldots, a_n$, as an n-ary parity or XOR constraint $a_1 \oplus \ldots \oplus a_n$ by multiple applications of $\oplus$. Since it is satisfied iff an odd number of atoms among $a_1, \ldots, a_n$ are true, we can simply refer it to as an *odd parity constraint*. Analogously, an *even parity constraint* is defined by $a_1 \oplus \ldots \oplus a_n \oplus \top$ as it is satisfied iff an even number of atoms among $a_1, \ldots, a_n$ hold. Then, e.g., $a_1 \oplus a_2 \oplus \top$ is satisfied iff none or both of a_1 and a_2 hold. Finally, there are four essential properties of an XOR, of which we can find associativity and commutativity. The other two properties are identity $a \oplus \bot \equiv a$ (resp. $a \oplus \top \equiv \top$), and self inverse where $a \oplus a \equiv \bot$. These constraints with a single atom are called *unary*.

2.5 The xorro System

Parity constraints have been recently accommodated in ASP as the basis of the system *xorro* [36], allowing the user to solve parity constraints on top of an ASP program.

xorro is a system that provides six alternatives to handle parity constraints into ASP solving through a standard syntax. On the one hand, *xorro* draws upon the advanced interfaces of *clingo* for integrating foreign constraints and corresponding forms of inference. On the other hand, *xorro* takes advantage of the sophisticated solving techniques developed in SAT for handling parity constraints. More precisely, *xorro* proposes two types of approaches, namely eager ones that rely on ASP encodings of parity constraints, and lazy ones using theory propagators within

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

clingo's Python interface [11]. [7] Table 1 summarizes the six implementations to handle parity constraints. The first three corresponds to the eager, and the last three to the lazy approaches.

The experiments from the paper evaluate the different approaches in view of their impact on solving performance while varying the number and density of parity constraints, compared against *clingo* solving time (without parity constraints). The results 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*.

To accommodate parity constraints in the input language, we rely on *clingo*'s theory language extension [11] following the common syntax of *aggregates* [44]:

```
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. [8] 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 $p(2) \oplus p(3) \oplus \top$, yielding the answer sets `{p(1)}` and `{p(1),p(2),p(3)}`. Currently, these constraints are interpreted as directives, and while partitioning the search space, they act as answer sets filters that do not satisfy the parity constraint in question. [9] 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.

[7] The distinction of eager and lazy approaches follows the methodology in Satisfiability modulo theories [1].
[8] In turn, multiple conditional terms within an aggregate are separated by semicolons.
[9] For now, parity constraints may not occur in the bodies nor the heads of rules.

3 Methodology and Approaches

This section describes and exemplifies our methodology by first introducing a running example using ASP as an overview to reverse the standard solving process. Then, we delve more into the underlying ASP encodings around parity constraints.

3.1 Running Example

Let us asume to have a system called *ProgramBuilder* which its core reiteratively calls *clingo*, and consists (among other features) in three stages. The first stage takes a set of answer sets and optionally SE properties as input, and delivers an intermediate representation. The second stage takes this intermediate representation to construct a starting propositional formula. The third and last stage takes this formula and proposes a new one strongly equivalent to the initial, according to the user needs.

The *ProgramBuilder* system benefits from the declarative approach of ASP, having a series of underlying programs comprises in a single one, called SPF that faithfully represents the entire system workflow. To describe this workflow, let us consider a very simple example with the intention to transmit our approach very clearly. Interested in finding a propositional formula of two variables p and q, such that it has $\{p\}$ and $\{q\}$ as unique answer sets, and discarding the empty set and $\{p, q\}$.

First Stage Departing from known answer sets as input, the system calls *clingo* and let it guess for a formula that satisfies the previous conditions. However, with these conditions, *clingo* finds over 300 different intermiediate representations (potential formulas) that satisfies the given input.

As mentioned before, the user can benefit from SE properties to delimit more the search. This means the user can straightforwardly specify in SPF the desired properties to satisfy. For example, the user can ask for a representation that satisfies *commutativity*, *associativity*, and *identity*. Calling again SPF coupled with the user-given properties, *clingo* encounters four intermediate representations.

Let us mention that these intermediate representations consist of a 3x3 matrix based on G_3 as shown in 2a and 2b. Now suppose we have two users, the first one, decides to refine more the search by adding another property, for example *idempotency*. Now *clingo* yields a single matrix, allowing the user to move to the second stage. The second user instead, asks *ProgramBuilder* to take two matrices M_1 and M_2 from the four remaining, postponing the decision, and moving also to the second stage.

Second Stage If the user has more than one matrix, he or she could enter into a dialog process until one solution is selected. However, the user could also keep the matrices and continue the workflow.

Suppose the user has two matrices M_1 and M_2, he or she wants to decide for one of them. To be more specific, let M_1 and M_2 be the matrices from tables 2a, and 2b respectively. We can see that both truth tables 2a, and 2b differ in a single value when

	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

(a) Truth table from M_1

	0	1	2
0	0	1	2
1	1	2	2
2	2	2	2

(b) Truth table from M_2

Table 2
M_1 and M_2 differing where both inputs equals to 1 in the logic of G_3.

both inputs are 1. [10] As mentioned before, these intermediate representations serve to construct initial propositional formulas. *ProgramBuilder* takes each matrix and constructs its corresponding formula. Let us recall that each formula is a disjunction of clauses, where each clause corresponds to the interpretations where the result equals to 2, and let the function F be responsible for the construction of the following formulas:

$$F_1 = F(M_1) = (p \wedge \neg q) \vee (q \wedge \neg p) \vee (p \wedge \neg\neg q) \vee (q \wedge \neg\neg p) \vee (p \wedge q)$$
$$F_2 = F(M_2) = (p \wedge \neg q) \vee (q \wedge \neg p) \vee (p \wedge \neg\neg q) \vee (q \wedge \neg\neg p) \vee (p \wedge q) \vee (\neg\neg p \wedge \neg\neg q)$$

ProgramBuilder can warn the user, that if you add another program Q, consisting of the rules $(p \leftarrow \neg q) \wedge (q \leftarrow \neg p)$ to both formulas F_1 and F_2, then, the computation of answer sets (represented by the function) $AS(F_1 \wedge Q) = \{\{p\}, \{q\}\}$, while $AS(F_2 \wedge Q) = \{\,\}$. In other words, for the first case, exist a single answer set $\{p, q\}$, opposed to the unsatisfiable second case. This means, that $F_1 \not\equiv_{SE} F_2$. It is relevant to mention that it is up to the user to pick one of them or to continue to the third stage.

Third Stage The system takes each formula and proposes a new strongly equivalent alternative. *ProgramBuilder* is equiped with an algebra of logical transformations (respecting SE), that can translate F_1 into a given normal-form. For this case, F_1 is reduced to the disjunction $p \vee q \vee (p \wedge q)$, that is SE to the constructed formula $p \vee q$. [11]

With the example above, we propose a first step methodology with the possibility to implement it into an interactive software that construct ASP programs through defined properties. Also, this software could include transformation modules to visualize multiple forms of constructed programs. Furthermore, this example serves to inspire the conception of a more general framework or concrete examples, as shown next.

3.2 Searching for Parity Aggregate Semantics

It is important to mention that this paper was driven by the search of the semantics of a parity aggregate through propositonal formulas (if exist) applying the definitions from Ferraris [23]. This aggregate comes as an improvement from the already existing parity constraints from *xorro*. In other words, we are searching for an operator of

[10] We present the tables for the reader and the sake of clarity. However, they could be irrelevant for the user. For the interested reader, the semantics of G_3 can be found in [38].

[11] For more details about the simplification, we refer to [25].

(at least) two entries that resembles a binary XOR respecting certain SE properties in ASP. [12] Let us motivate again with the definition of the problem.

Problem definition. Given (an incomplete set of) answer sets, and optionally strong equivalent properties as input, we want to test an existing intermediate representation or search for a new one, to construct a single or several propositional formulas from (and to be lately used by) ASP.

To do so, we could either add our intermediate representation or follow the standard *guess-and-check* paradigm of ASP. In both cases, solution candidates are tested for feasibility with the possibility of yielding none, one, or multiple answer sets. Typically, these answer sets serve as the solutions of an encoded program, but for our purposes, they are interpretations in G_3 which allow us to construct formulas. Before delving into the three stages, let us explain that we have two types of answer sets (due to our *meta-programming* approach), the answer sets given as input, and the answer sets as intermediate representations. From now on, we easily differentiate them as $answer\_set(s)_{input}$ and $answer\_set(s)_{output}$ respectively, and we use them interchangeably.

Our implementation of the core problem into ASP follows the common practices of ASP, by separately provide an instance and an encoding. As stated above in the definition of the problem, the instance corresponds to $answer\_sets_{input}$ and SE properties, and the encoding consists of means to prove the existence or not of a propositional formula.

Once more, we start from known $answer\_sets_{input}$ $\{p\}$ and $\{q\}$, and discarding again the empty set and $\{p, q\}$, using the same variables p and q. We represent each input answer set with the atom `answer_set` having a string value as its argument, as shown in Listing 1.

```
1  :- not answer_set("p").
2  :- not answer_set("q").
3  :- answer_set("").
4  :- answer_set("p q").
```

Listing 1: Answer sets as part of the instance (`answer_sets.lp`).

Since the requirements are clear beforehand, and as part of the first stage, we can represent the four essential properties from the classical logic XOR as part of our input. The properties are the same from Section 2.4, *commutativity*, *associativity*, *self inverse*, and *identity*. [13] To see these SE properties in the context of ASP, we refer to Listing 2.

```
1  %% Commutativity :  X xor Y = Y xor X
2  :- op(X,Y,R1), op(Y,X,R2), R1!=R2.

4  %% Associativity : (X xor Y) xor Z = X xor (Y xor Z)
5  left(X,Y,Z,R)  :- op(X,Y,W1), op(W1,Z,R). %% Left
6  right(X,Y,Z,R) :- op(Y,Z,W1), op(X,W1,R). %% Right
7  :- left(X,Y,Z,R1), right(X,Y,Z,R2), R1 != R2.

9  %% Self Inverse  :  X xor X = 0
10 :- op(X,X,R), R!=0.
```

[12] This paper is not intended to propose a complete description of a parity aggregate and its semantics. We intend only to use the methodology as a systematic proof to find propositional formulas that satisfies properties from the classical XOR operator.

[13] For the identity property, a variable p XORed with $\bot$, equals to the double negation of the input. For instance, $p \oplus \bot$ equals $\neg\neg p$. This is represented in ASP as the constraint: `:- not p.`

```
%% Identity      :  X xor 0 = not not X
:- op(X,0,Y), neg(X,X1), neg(X1,Z), value(Y), Y != Z.
```

Listing 2: Essential SE properties of the classical XOR operator (`xor_strong.lp`).

From the code above, we represent each property as an integrity constraint, where the atom `op(X,Y,R)` corresponds to the desired operator of two arguments (variables) X and Y and its result R. Let us allow to get ahead, and mention that this atom is part of the $answer\_sets_{output}$ or intermediate representation. Before we move to the second stage, we describe the encodings that yield the intermediate representations.

The encoding consists of four parts, the test of an existing intermediate representation or the guess of a new one, the definition of the logical operators, the theory completion, and G_3 persistency properties.

Of course, it was a completely natural and straightforward situation, to test the classical XOR formula as a first attempt, rather than to ask directly for an intermediate representation. That is, we first encode the classical XOR formula $(p \vee q) \wedge (\neg p \vee \neg q)$ into a rule as shown in Listing 3.

```
op(X,Y,Z) :- or(X,Y,R1), neg(X,X1),neg(Y,Y1),or(X1,Y1,R2), and(R1,R2,Z).
```

Listing 3: Encoding of the classical XOR formula (`classical_xor.lp`).

In this encoding, we first represent the disjunction of the two inputs X and Y, and then, we negate both inputs for the second disjunction. Lastly, we conjunct both results.

Also, instead of coming up with different intermediate representations, encode them, and test their satisfiability, we can replace the rule from Listing 3, with the choice rule from Listing 4, allowing *clingo* guess for an operator `op(X,Y,R)`, from any two values X and Y, resulting in R, which is no other than a choice rule with both boundaries set to one.

```
1 { op(X,Y,Z) : value(Z) } 1 :- value(X), value(Y).
```

Listing 4: Guess formula via an interpretation in G_3 (`guess_formula.lp`).

The second part of the encoding is the definition of the logical operators in G_3 *and*, *or*, *negation*, and *implication*, as shown in Listing 5.

```
value(0..2). %% G3 values

and(X,X,X) :- value(X).
and(X,Y,X) :- value(X), value(Y), X<Y.
and(X,Y,Y) :- value(X), value(Y), Y<X.

or(X,X,X) :- value(X).
or(X,Y,X) :- value(X), value(Y), Y<X.
or(X,Y,Y) :- value(X), value(Y), X<Y.

neg(0,2).  neg(2,0).  neg(1,0).

implication(X,Y,2) :- value(X), value(Y), X <= Y.
implication(X,Y,Y) :- value(X), value(Y), X > Y.
```

Listing 5: G_3 values and logical operators (`logical_operators.lp`).

To find or test $answer\_sets_{ouput}$, we need to characterize them in terms of what

we call a theory completion, as shown in Listing 6. [14]

```
1  completion(0,X,Y,R):- neg(X,X1), neg(Y ,Y1), and(X1,Y1,R).
2  completion(1,X,Y,R):- neg(X,X1), neg(X1,X2), neg(Y,Y1), and(X2,Y1,R).
3  completion(2,X,Y,R):- neg(Y,Y1), neg(Y1,Y2), neg(X,X1), and(X1,Y2,R).
4  completion(3,X,Y,R):- neg(Y,Y1), neg(Y1,Y2), neg(X,X1), neg(X1,X2), and(X2,
       Y2,R).

6  belongs(1,p).  belongs(2,q).  belongs(3,p).  belongs(3,q).

8  code(0,"").  code(1,"p").  code(2,"q").  code(3,"p q").

10 completion_asp(A_ID,X,Y,R) :- op(X,Y,Z), completion(A_ID,X,Y,C), and(Z,C,R)
       .

12 consistent(A_ID):-completion_asp(A_ID,X,Y,R),value(R),R>0.
13 incomplete(A_ID):-belongs(A_ID,x), completion_asp(A_ID,X,Y,Z), implication(
       Z,X,R), R<2.
14 incomplete(A_ID):-belongs(A_ID,y), completion_asp(A_ID,X,Y,Z), implication(
       Z,Y,R), R<2.

16 answer_set(S) :- consistent(A_ID), not incomplete(A_ID), code(A_ID,S).
```

Listing 6: Theory completion for answer sets (`theory_completion.lp`).

Describing an overview of the main function of this code, the first four rules from Listing 6, captures the completions needed for all possible answer sets (related to $answer\_set_{input}$) concerning our inputs p and q. Then, the facts in line 6, display the correspondence between the constants p and q with all the possible answer sets, followed by (facts) mappings into string representations in line 8. Line 10 forms the completion concerning the operator. Then, the completion must be consistent (line 12), and we define what incompleteness is (lines 13 and 14). Lastly, line 16, derives the corresponding answer sets in string representation via their correlated code. These answer sets must satisfy consistency and completeness, as well as the $answer\_sets_{input}$ (Listing 1).

Finally, we need to guarantee G_3 persistence properties [29,25]. They are displayed in Listing 7. Here, line 1 states that it is not possible that in case there exist an interpretation {1,0,2}, then exist another intepretation with inputs 2 and 0, that evaluates to any other value different than 2. Line 2 describes the commutated property, and line 3 states that, is not possible that an interpretation resulting in 1, comes from inputs different than 1.

```
1  :- op(1,0,2), op(2,0,X), X != 2.
2  :- op(0,1,2), op(0,2,X), X != 2.
3  :- op(X,Y,1), X != 1, Y != 1.
```

Listing 7: G_3 persistence (`g3_persistence.lp`).

Solving both, the instance, and the encoding produces the intermediate representations to move to the second stage. However, for either both cases (using the classical XOR formula or guessing for one), there are no $answer\_sets_{output}$. This means it is not possible to represent an XOR operator as a two arguments function in ASP, that besides, satisfy all four properties. Despite the negative solution where there is no formula to construct, it is positive in the sense that this methodology can save time, money, and resources from a software engineering perspective. Also, this fits perfectly into iterative software engineering methodologies, taking the user

[14] For the sake of clarity, we fix this encoding concerning the exemplary signature $\{p, q\}$. However, it is possible to generate this encoding for a given signature.

Interpretation	Clause			
$\{0,1,2\}$	$\neg$	$p \wedge$	$\neg\neg$	q
$\{0,2,2\}$	$\neg$	$p \wedge$		q
$\{1,0,2\}$	$\neg\neg$	$p \wedge$	$\neg$	q
$\{2,0,2\}$		$p \wedge$	$\neg$	q

Table 3
Resulting clauses from Matrix 2.

back to the initial or design stage, wondering about the requirements.

On the other hand, and following our XOR motivation, we also question ourselves if we can find a formula that semantically behaves as a parity constraint as the ones used in *xorro* [36]. That is, we are searching for a constraint formula that discards candidate answer sets from an independent generation process, that do not satisfy both, the $answer\_sets_{input}$ and the SE properties. Thus, the XOR formula now consists of two parts, the generate section in conjunction with the test or constraint. The generation process consists of the conjunction of clauses of the form $x \vee \neg x$, for every variable x. Listing 8 shows the encoding for the generation of all answer sets concerning any two values X and Y into the predicate `xor_g/3`. The test is modeled as a choice rule like the one from Listing 4 but with the new predicate `xor_t/3`. Combining both, it results in the rule
`op(X,Y,Z) :- xor_g(X,Y,Z1), xor_t(X,Y,Z2), and(Z1,Z2,Z).`
With this approach, it is relevant to mention that the SE properties must evaluate only the test part of the formula, and not the whole expression or the generate.

```
1  xor_g(X,Y,Z) :- neg(X,X1), neg(Y,Y1), or(X,X1,Z1), or(Y,Y1,Z2), and(Z1,Z2,Z
       ).
```

Listing 8: The generation of candidate answer sets (`generate.lp`).

After asking exhaustively for all possible intermediate representations, *clingo* found a single $answer\_set_{output}$. This means, there is only one alternative to represent an XOR as a constraint in ASP satisfying the aforementioned properties as shown in matrix 2. With this intermediate representation as a matrix of the form of M_1 or M_2 (from the running example), we can move to the second stage.

```
xor_t(0,0,0), xor_t(0,1,2), xor_t(0,2,2)
xor_t(1,0,2), xor_t(1,1,0), xor_t(1,2,0)        (2)
xor_t(2,0,2), xor_t(2,1,0), xor_t(2,2,0)
```

Since we have only one representation, we construct the formula F_{xor_t}, which gives the following clauses from Table 3, taking the specification shown in Eq. 1. This results in the initial propositional formula:

$$F_{xor\_t} = (\neg p \wedge \neg\neg q) \vee (\neg p \wedge q) \vee (\neg\neg p \wedge \neg q) \vee (p \wedge \neg q)$$

Lastly, the third stage proposes a transformation for F_{xor_t}. For instance, a resulting formula or a simplification from F_{xor_t} could be $(\neg\neg p \vee \neg\neg q) \wedge (\neg p \vee \neg q)$.

Viewing it from the opposite direction, we can reverse the presented method by given a propositional formula and let *clingo* not only test it but search for the answer sets. For example, let us encode the reduced F_{xor_t} formula as:

```
xor_t(X,Y,Z2) :- or(X,Y,R1), neg(X,X1), neg(Y,Y1), or(X1,Y1,R2),
and(R1,R2,Z), neg(Z,Z1), neg(Z1,Z2).
```

This formula replaces the code from Listings 1, 4, and 3 and reuse the aforementioned encoding, logical operators (Listings 5), theory completion (Listings 6), and G_3 persistency properties (Listings 7). Therefore, *clingo* yields the answer sets $\{p\}$ and $\{q\}$, as well as the same G_3 representation shown in matrix 2. This means that the founded formula semantically behaves as a parity constraint used in *xorro*.

Finally, it is worth mentioning that currently, we have an initial and very basic implementation using Python and *clingo*. For more details, go to https://github.com/flavioeverardo/Propositional-Formula-Builder-PFB.

Also, a full, and a ready to use encoding combining all the listings from the paper is available at

https://github.com/flavioeverardo/Propositional-Formula-Builder-PFB/blob/master/lp/full_xor_encoding.lp.

4 Discussion

The automatic generation of solutions for declaratively specified search-problems is one of the most successful areas of artificial intelligence [4], where Answer Set Programming highlights due to its full support on a compact representation of search problems. Additionally, the need for automatic source code optimization, and software engineering tools and methodologies into ASP come hand in hand. Inspired (among others) by circumstances where a problem encoded into an ASP might not have the desired solving performance compared to an equivalent representation.

Motivated by these needs, we presented a preliminary developing process towards a methodology to implement ASP programs, following existing methods. We captured this developing process into an initial prototype consisting of ASP encodings, that reverses the standard solving workflow towards an exhaustive search for propositional formulas, all within ASP. The resultant formula(s) must satisfy strong equivalent (SE) properties as well as known or desired answer sets.

This paper is an extended version from [6], which allows us to delve a little more in the context of finding a semantics for a parity aggregate as an improvement from the parity constraints used in *xorro*. Our approach, let us not only test propositional formulas but also ask for their existence through ASP solving, paving the road towards a benchmarking procedure of ASP programs, to find an optimal representation.

For future work, there is too much to do. First of all, we plan to continue the development of a fully-integrated software concerning the proposed methodology, including the tools for reconstructing more complex formulas. This initial prototype

uses ASP in its whole, as it is conceived thanks to high-level interfaces, sophisticated algorithms for grounding and solving, including search heuristics and learning techniques based on nogoods, among others. Hence, this initiative constructs propositional formulas. However, it is far from fully equipped software. One possible extension could be the construction of not only propositional formulas, but *non-ground* ASP programs. That is, including variables. Then, we could benefit from tools like *anthem* [37] for verifying SE programs in the input language of *gringo*. The advantage is that we can safely replace a piece of knowledge representation with another regardless of the context. In other words, we can safely change code without modifying the semantics of the program.

In terms of software engineering, and to the best of our knowledge, [14] is the only approach describing a standard software engineering process consisting of the development and the design of ASP programs which has been tested in an industrial context. However, both our method and the prototype could benefit from several other techniques from the ASP community which has a relationship with software engineering, such as Inductive Logic Programming (ILP) [15,16], Procedural Content Generation (PCG) [17], ASP Debbuggers [18] (including Meta-Programming [19]), and an IDE for ASP called ASPIDE [20].

References

[1] Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere et al. [2], chap. 26, pp. 825–885.

[2] Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009).

[3] 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).

[4] Schaub, T., and Woltran, S.: Answer set programming unleashed!. KI-Künstliche Intelligenz, 32(2-3), 105-108, (2018).

[5] Lifschitz, V.: Answer set planning. In International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 373–374. Springer, Berlin, Heidelberg (1999).

[6] Everardo, F., Osorio, M.: Towards an Answer Set Programming Methodology for Constructing Programs Following a Semi-Automatic Approach Accepted to apper in the Twelve Latin American Workshop on New Methods of Reasoning 2019 (LANMR 2019).

[7] Gebser, M., Kaufmann, B., and Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence, 187, 52-89 (2012).

[8] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T.: Answer set solving in practice. Synthesis lectures on artificial intelligence and machine learning, 6(3), 1-238 (2012).

[9] Gebser, M., Kaminski, R., König, A., and Schaub, T.: Advances in gringo series 3. In International Conference on Logic Programming and Nonmonotonic Reasoning (pp. 345-351). Springer, Berlin, Heidelberg (2011).

[10] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T.: Clingo = ASP + Control: Preliminary Report. CoRR, abs/1405.3694 (2014).

[11] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P.: Theory solving made easy with clingo 5. In Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016).

[12] Gebser, M., Kaminski, R., and Schaub, T.: Complex optimization in answer set programming. Theory and Practice of Logic Programming, 11(4-5), 821-839 (2011).

[13] Eiter, T., Ianni, G., and Krennwallner, T.: Answer set programming: A primer. In Reasoning Web International Summer School (pp. 40-110). Springer, Berlin, Heidelberg (2009, August).

[14] Falkner, A., Friedrich, G., Schekotihin, K., Taupe, R., and Teppan, E. C.: Industrial applications of answer set programming. KI-Künstliche Intelligenz, 32(2-3), 165-176, (2018).

[15] Corapi, D., Russo, A., and Lupu, E.: Inductive logic programming in answer set programming. In International Conference on Inductive Logic Programming (pp. 91-97). Springer (2011).

[16] Law, M., Russo, A., and Broda, K.: Inductive learning of answer set programs. In European Workshop on Logics in Artificial Intelligence (pp. 311-325). Springer, Cham (2014).

[17] Smith, A. M., and Mateas, M.: Answer set programming for procedural content generation: A design space approach. IEEE Transactions on Computational Intelligence and AI in Games, 3(3), 187-200 (2011).

[18] Brain, M., and De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In Answer Set Programming (2005).

[19] Gebser, M., Pührer, J., Schaub, T., and Tompits, H.: A meta-programming technique for debugging answer-set programs. In AAAI (Vol. 8, pp. 448-453) (2008).

[20] Febbraro, O., Reale, K., and Ricca, F.: ASPIDE: Integrated development environment for answer set programming. In International Conference on Logic Programming and Nonmonotonic Reasoning (pp. 317-330). Springer, Berlin, Heidelberg (2011).

[21] Lifschitz, V., Pearce, D., and Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic (TOCL), 2(4), 526-541, (2001).

[22] Turner, H.: Strong equivalence made easy: nested expressions and weight constraints. Theory and Practice of Logic Programming, 3(4+ 5), 609-622 (2003).

[23] Ferraris, P.: Answer Sets for Propositional Theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05). Lecture Notes in Artificial Intelligence, vol. 3662, pp. 119–131. Springer-Verlag (2005).

[24] Cabalar, P., and Ferraris, P.: Propositional theories are strongly equivalent to logic programs. Theory and Practice of Logic Programming, 7(6), 745-759 (2007).

[25] Aguado, F., Cabalar, P., Fandinno, J., Pearce, D., Pérez, G., and Vidal, C.: Forgetting auxiliary atoms in forks. Artificial Intelligence, 275, 575-601, (2019).

[26] Project Management Institute.: A Guide to the Project Management Body of Knowledge (PMBOK Guide)–Sixth Edition (2017).

[27] Heyting A.: Die formalen Regeln der intuitionistischen Logik, Sitz. Berlin 42-56 (1930).

[28] Gödel, K,: Zum intuitionistischen Aussagenkalkül, Anzeiger der Akademie der Wissenschaften in Wien 69 65-66; reprinted in em Kurt Gödel, Collected Works, Volume 1, OUP, (1986).

[29] Osorio, M., Navarro, J. A., and Arrazola, J.: Equivalence in answer set programming. In International Workshop on Logic-Based Program Synthesis and Transformation (pp. 57-75). Springer, Berlin, Heidelberg (2001).

[30] Osorio, M., Navarro, J. A., and Arrazola, J.: Applications of intuitionistic logic in answer set programming. Theory and Practice of Logic Programming, 4(3), 325-354 (2004).

[31] Osorio, M., Navarro, J. A., and Arrazola, J.: Safe beliefs for propositional theories. Annals of Pure and Applied Logic, 134(1), 63-82 (2005).

[32] Pearce, D.: A new logical characterisation of stable models and answer sets. In International Workshop on Non-monotonic Extensions of Logic Programming (pp. 57-70). Springer (1996).

[33] Navarro, J. A.: Answer Sets through G3 Logic. In ESSLLI Student Session p. 181 (2002).

[34] Cabalar, P., Pearce, D., and Valverde, A.: Reducing propositional theories in equilibrium logic to logic programs. In Portuguese Conference on Artificial Intelligence (pp. 4-17). Springer, Berlin, Heidelberg (2005).

[35] Cabalar, P., Pearce, D., and Valverde, A.: Minimal logic programs. In International Conference on Logic Programming (pp. 104-118). Springer, Berlin, Heidelberg (2007).

[36] 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).

[37] Lifschitz, V., Lühne, P., and Schaub, T.: Verifying Strong Equivalence of Programs in the Input Language of gringo. In International Conference on Logic Programming and Nonmonotonic Reasoning (pp. 270-283). Springer, Cham (2019).

[38] Ultlog, M.: Calculi for the Gödel Logic (2001).

[39] Chakraborty, S., Meel, K., Vardi, M.: A scalable and nearly uniform generator of SAT witnesses. In: Sharygina, N., Veith, H. (eds.) Proceedings of the Twenty-fifth International Conference on Computer Aided Verification (CAV'13). Lecture Notes in Computer Science, vol. 8044, pp. 608–623. Springer-Verlag (2013).

[40] Chakraborty, S., Meel, K., Vardi, M.: A scalable approximate model counter. In: Schulte, C. (ed.) Proceedings of the Nineteenth International Conference on Principles and Practice of Constraint Programming (CP'13). Lecture Notes in Computer Science, vol. 8124, pp. 200–216. Springer-Verlag (2013).

[41] Gomes, C., Hoffmann, J., Sabharwal, A., Selman, B.: Short XORs for model counting: from theory to practice. In: Marques-Silva, J., Sakallah, K. (eds.) Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07). Lecture Notes in Computer Science, vol. 4501, pp. 100–106. Springer-Verlag (2007).

[42] Gomes, C., Sabharwal, A., Selman, B.: Near-uniform sampling of combinatorial spaces using XOR constraints. In: Schölkopf, B., Platt, J., Hofmann, T. (eds.) Proceedings of the 20th Annual Conference on Neural Information Processing Systems (NIPS'06). pp. 481–488. MIT Press (2007).

[43] Soos, M., Meel, K.: Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In: Van Hentenryck, P., Zhou, Z. (eds.) Proceedings of the Thirty-third National Conference on Artificial Intelligence (AAAI'19). AAAI Press (2019).

[44] Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**(4-5), 449–463 (2015).

[45] Laitinen, T.: Extending SAT Solver with Parity Reasoning. Aalto University (2014).