Towards an Answer Set Programming Methodology for Constructing Programs Following a Semi-Automatic Approach

Flavio Everardo[1] and Mauricio Osorio[2]

[1] University of Potsdam, Germany
flavio.everardo@cs.uni-potsdam.de
[2] Universidad de las Americas Puebla, Mexico
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, paving the road towards a benchmarking procedure of ASP programs.

1 Introduction

The automatic generation of solutions for declaratively specified search-problems is one of the most successful areas of artificial intelligence [1], where Answer Set Programming (ASP; [2]) highlights due to its full support on a compact representation of search problems. ASP 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 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* [3] demonstrating its competitiveness and versatility, by winning first places at various solver contests since 2011 (eg. ASP, CASC, MISC, PB, and SAT competitions). [3]

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

clasp, combined with the grounder *gringo* [5], composes *clingo* [6], an ASP system to ground and solve logic programs. For the reader interested in learning more about ASP, including theoretical works, implementations, and applications, see [9,7,4,6].

Despite the success of ASP in both academic and industry, [4] 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 [1].

To the best of our knowledge, [10] is the only approach describing a standard software engineering process consisting of the development and the design of ASP programs in an industrial context. Other works in ASP that have some relationship with software engineering, concerns Inductive Logic Programming (ILP) [11,12], Procedural Content Generation (PCG) [13], ASP Debbuggers [14] (including Meta-Programming [15]), and an IDE for ASP called ASPIDE [16].

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 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 [10]. Second, we present ASP encodings that fall under the category of *meta-programs* [8] serving 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, exhaustively reconstruct ASP programs if they exist, following the approaches from [20,21], paving the road towards a benchmarking procedure of ASP programs, to find an optimal representation. Informally, strong equivalence (SE) means 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.

To motivate this paper, let us set the context of the intended process, and let us illustrate a running example using ASP as an overview to reverse the standard solving process. For more fine-grained details including ASP codifications, we refer to Section 3.

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 the answer sets and possible strong equivalent 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 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

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

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 the 3-valued logic of G_3. 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 1a, and 1b respectively. We can see that both truth tables 1a, and 1b differ in a single value when both inputs are 1. [5] As mentioned before, these intermediate representations serve

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

to construct initial propositional formulas. $ProgramBuilder$ takes each matrix and constructs its corresponding formula. Let us state 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, $AS(F_1 \wedge Q) =$

[5] 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 [34].

$\{\{p\},\{q\}\}$, while $AS(F_2 \wedge Q) = \{\,\}$. In other words, for the first case, we have the single answer set $\{p, q\}$, and for the second case, there are no answer sets (unsatisfiable). This means, that F_1 and F_2 are not strongly equivalent. 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 translated into to the logical disjunction $p \vee q$. 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. As mentioned above, this software could include transformation modules to visualize the constructed programs in multiple forms. We present this toy example on purpose to make it easy to follow. Nevertheless, this example inspires the conception of a more general framework.

The remainder of the paper is structured as follows: Section 2 informally introduces ASP, followed by the formal definition of strong equivalence. Then, we focus on the introduction of non-standard concepts needed in this paper, like the approach to construct formulas from an interpretation in the 3-valued logic of G_3. We also mention the straightforward relationship with the logic of of Here-and-There (HT). We close this section with the best practices for designing and developing ASP programs. Section 3 describes our methodology by complementing the running example, illustrating with more complex examples as well as ASP programs. Finally, we discuss the conclusions of 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) [24], and its relationship with the logic of Here-and-There (HT) [23], among others. Lastly, we recapitulate the design and development process of ASP programs from [10].

2.1 Strong Equivalence

The term *Strong Equivalence* [17], concerning ASP programs, means that, having two programs (formulas) F_1 and F_2, F_1 is strongly equivalent to F_2 if F_1 is equivalent to F_2 in the Gödel's 3-valued logic (G_3), which is equivalent to the logic of *Here-and-There* (HT). Also, via the *reduct* [27], F_1 is strongly equivalent to F_2 if for each set X of atoms both *reducts* F_1^X and F_2^X are equivalent in classical logic [18,19]. 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 [20,21]. Yet, it is also possible to use G_3 logic, which it is equivalent to HT, and the relationship is straightforward. [6]

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 G_3-interpretation I (as shown in Tables 1a and 1b), we apply the following specification or clause C from [21]. [7] To create a clause, we apply the formula below whenever an interpretation equals to 2. A more detailed example is shown in Table 2 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, as shown in F_1 from the running example. Lastly, the formula can be simplified according to (but not necessarily all) [27,30,31] Taking back the running example, the formula F_1 is reduced to the disjunction $p \vee q \vee (p \wedge q)$, that is strongly equivalent to the constructed formula $p \vee q$. [8]

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 [26]. 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 [10] 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) [22]. We recapitulate the six areas and let us point out the intersection with the aforemention stages from our methodology.

1. **Identify the needs** Find opportunities where ASP is stronger than conventional methods. Define and document the application requirements properly (first stage).
2. **Design a valid specification of the problem** Implement an ASP specification of the core problem with small instances for testing. Take advantages of ASP which allows interactive problem refinement and tuning (second stage with support from the first stage).

[6] We only mention the needed concepts from the logics of G_3 and HT. For more information, we may refer the reader to [28,29].

[7] The original formula is in the context of HT. To be consistent, we adapted for the G_3.

[8] For more details about the simplification, we refer to [21].

3. **Performance engineering** Explore alternatives of ASP program implementations (as shown in the third stage with the support of the other two stages), and evaluate their performance considering "real-world" size instances. [9] For a prototyping process, like our methodology, we focus more on readability rather than performance.
4. **Integrate into the existing environment** Choose the best ASP program alternative from the feasibility study and implement a clean ASP program which processes the transformed data. Evaluate if is possible to use incremental solving, and consider the manipulation of answer sets. Design interfaces and implement a complete and efficient transformation from legacy input data to ASP and back. It is now possible to integrate ASP solvers like *clingo*, into legacy systems in a more natural way due to a complete API in languages like Python or C++.
5. **Testing and debugging** Ensure high-quality via automated tests, and debugging of ASP programs if applicable. For instance, ASP Debbuggers like [14,15].
6. **Maintenance** Focus on a well-defined structure of the program, and benefit from ASP's modularity for further adaptions.

Also, [10] 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 [10], 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.

3 Methodology and Approaches

This section describes and exemplifies our methodology, delving more into the underlying ASP encodings. That is, we retake our running example, and decompose the three stages with more complex examples. Let us motivate again with the definition of the problem.

Problem definition. Given (an incomplete set of) answer sets, and possible strong equivalent properties as input, search and construct a single or several propositional formulas, hence, an ASP program satisfying these conditions. To do so, we follow the standard *guess-and-check* paradigm of ASP where 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

[9] 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.

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 strong equivalent properties. On the other hand, the encoding consists of means to prove the existence or the lack of a propositional formula. Before addressing the three stages, let us illustrate our workflow with another example. Originally, a motivation behind this work was the question about the existence of an interpretation, that satisfies the four essential properties from the exclusive disjunction (XOR).

Let us request a propositional formula, keeping the same variables p and q, and the same unique $answer\_sets_{input}$ $\{p\}$ and $\{q\}$, while discarding again the empty set and $\{p, q\}$. 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").
:- not answer_set("q").
:- answer_set("").
:- answer_set("p q").
```

Listing 1.1: 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, where two of them are *commutativity* and *associativity*. The other two properties are *self inverse*, meaning that any input XORed with itself is false, and *identity*, where an input XORed by false, yields the double negation of the entry. [10] To see these properties in the context of an instance, we refer to Listing 1.2.

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

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

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

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

Listing 1.2: 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

[10] By letting us expressing that a variable p XORed with a false constant as $p \oplus \bot$, equals $\neg\neg p$. This is represented in ASP as a constraint of the form: `:- not p.`

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 guessing of the intermediate representation, the definition of the logical operators, the theory completion, and G_3 persistency properties. Let us start with the intermediate representation guessing, which is no other than a choice rule with both boundaries set to one, asking for an operator `op(X,Y,R)`, from any to values X and Y, resulting in R (Listing 1.3).

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

Listing 1.3: Guess formula via an interpretation in G_3 (`guess_formula.lp`).

The second part is the definition of the logical operators in G_3, shown in Listing 1.4.

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

This encoding consisting of the operators *and*, *or*, *negation*, and *implication*, serves twofold. It works for the generation of intermediate representations or to compute the answer sets if given an initial formula. [11] To find intermediate representations ($answer\_sets_{ouput}$), we need to characterize them in terms of what we call a theory completion, as shown in Listing 1.5. [12]

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

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

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

[11] We discuss this point at the end of the section.

[12] 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.

```
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,p), completion_asp(A_ID,X,Y
        ,Z), implication(Z,X,R), R<2.
14  incomplete(A_ID):-belongs(A_ID,q), 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 1.5: Theory completion for answer sets (`theory_completion.lp`).

Describing an overview of the main function of this code, the first four rules from Listing 1.5, 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 1.1).

Finally, we need to guarantee G_3 persistence properties [25,21]. They are displayed in Listing 1.6. 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 1.6: 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 this particular example, there are no $answer\_sets_{output}$. This means it is not possible to represent an XOR operator as a function of two arguments in ASP, that aside, 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. Also, this fits perfectly into iterative software engineering methodologies, taking the user 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* [32]. That is, we are searching for a constraint formula that discards candidate answer sets from an independent generation process. [13] We affirmatively answer this

[13] The generation process for every variable x is represented as $x \vee \neg x$.

question, and this comes after an exhaustive search. In other words, we ask *clingo* for all possible intermediate representations, and clingo found a single $answer\_set_{output}$. This means that there is only one possibility to represent an XOR as a constraint in ASP satisfying the aforementioned properties. By these means, we can confirm that the founded formula semantically behaves as the parity constraints used in *xorro*.

```
op(0,0,0), op(0,1,2), op(0,2,2)
op(1,0,2), op(1,1,0), op(1,2,0)        (2)
op(2,0,2), op(2,1,0), op(2,2,0)
```

With this intermediate representation (2) as a matrix of the form of M_1 or M_2 (from the running example), we can move to the second stage.

For this example, we do not have more than one representation, so we do not have anything else to compare. Hence, the constructed formula, namely F_{xor} gives the following clauses (2), taking the specification shown in 1. This results in the initial propositional formula:

Table 2: Resulting clauses after the interpretation in G_3.

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

$$F_{xor} = (\neg p \wedge \neg\neg q) \vee (\neg p \wedge q) \vee (\neg\neg p \wedge \neg q) \vee (p \wedge \neg q) \tag{3}$$

Lastly, the third stage proposes a transformation for F_{xor}. For instance, a resulting formula could be $(\neg\neg p \vee \neg\neg q) \wedge (\neg p \vee \neg q)$. Nevertheless, it is possible to reverse the presented method by given a propositional formula and let *clingo* search for the answer sets. For example, let our instance be the same F_{xor} formula using the logical operators from Listing 1.4, as:

```
op(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 1.1, and 1.3, and reuse the aforementioned encoding, logical operators (Listings 1.4), theory completion (Listings 1.5), and G_3 persistency properties (Listings 1.6). Therefore, generating candidate answer sets over p and q, F_{xor} discards the empty set and $\{p, q\}$.

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*.

4 Discussion

Motivated by the need for automatic source code optimization, and the inclusion of software engineering into ASP, 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 properties as well as known answer sets.

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* [33] to verifying Strong Equivalence of ASP programs in the input language of *gringo*.

In terms of software engineering, both our method and the prototype could benefit from several other techniques from the ASP community, fitting perfectly with the uncovered steps (4,5, and 6) from [10] as well as into the OOASP approach. Some of these techniques, includes Inductive Logic Programming, debugging, Graphical User Interfaces encouraged by ASPIDE, to name a few.

References

1. Schaub, T., and Woltran, S.: Answer set programming unleashed!. KI-Künstliche Intelligenz, 32(2-3), 105-108, (2018).
2. Lifschitz, V.: Answer set planning. In International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 373–374. Springer, Berlin, Heidelberg (1999).
3. Gebser, M., Kaufmann, B., and Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence, 187, 52-89 (2012).
4. 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).
5. 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).
6. Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T.: Clingo = ASP + Control: Preliminary Report. CoRR, abs/1405.3694 (2014).
7. 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). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016).
8. 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).
9. Kaminski, R., Schaub, T., and Wanko, P.: A tutorial on hybrid answer set solving with clingo. In Reasoning Web International Summer School (pp. 167-203). Springer, Cham (2017).
10. 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).
11. 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, Berlin, Heidelberg (2011).
12. 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).
13. 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).

14. Brain, M., and De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In Answer Set Programming (2005).
15. 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).
16. 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).
17. Lifschitz, V., Pearce, D., and Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic (TOCL), 2(4), 526-541, (2001).
18. Turner, H.: Strong equivalence made easy: nested expressions and weight constraints. Theory and Practice of Logic Programming, 3(4+ 5), 609-622 (2003).
19. 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)
20. Cabalar, P., and Ferraris, P.: Propositional theories are strongly equivalent to logic programs. Theory and Practice of Logic Programming, 7(6), 745-759 (2007).
21. Aguado, F., Cabalar, P., Fandinno, J., Pearce, D., Pérez, G., and Vidal, C.: Forgetting auxiliary atoms in forks. Artificial Intelligence, 275, 575-601, (2019).
22. Project Management Institute.: A Guide to the Project Management Body of Knowledge (PMBOK Guide)–Sixth Edition (2017).
23. Heyting A.: Die formalen Regeln der intuitionistischen Logik, Sitz. Berlin 42-56 (1930).
24. 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).
25. 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).
26. 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).
27. Osorio, M., Navarro, J. A., and Arrazola, J.: Safe beliefs for propositional theories. Annals of Pure and Applied Logic, 134(1), 63-82 (2005).
28. 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, Berlin, Heidelberg (1996).
29. Navarro, J. A.: Answer Sets through G3 Logic. In ESSLLI Student Session p. 181 (2002).
30. 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).
31. Cabalar, P., Pearce, D., and Valverde, A.: Minimal logic programs. In International Conference on Logic Programming (pp. 104-118). Springer, Berlin, Heidelberg (2007).
32. 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)
33. 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) .
34. Ultlog, M.: Calculi for the Gödel Logic (2001).