Achievements in Answer Set Programming

Vladimir Lifschitz

November 13, 2017

Abstract

This paper describes an approach to the methodology of answer set programming that can facilitate the design of encodings that are easy to understand and provably correct. Under this approach, after appending a rule or a small group of rules to the emerging program we include a comment that states what has been "achieved" so far. This strategy allows us to set out our understanding of the design of the program by describing the roles of small parts of the program in a mathematically precise way.

1 Introduction

This paper describes an approach to the methodology of answer set programming [Marek and Truszczynski, 1999, Niemelä, 1999] that can facilitate the design of encodings that are easy to understand and provably correct. Under this approach, after appending a rule or a small group of rules to the emerging program, the programmer would include a comment that states what has been "achieved" so far, in a certain precise sense.

Consider, for instance, the following solution to the 8 queens problem, adapted from [Gebser *et al.*, 2012, Section 3.2].[1]

```
1 % Program 8Queens
2
3 row(1..8).
4 col(1..8).
5 8 { queen(I,J) : col(I), row(J) } 8.
6 :- queen(I,J), queen(I,JJ), J!=JJ.
7 :- queen(I,J), queen(II,J), I!=II.
8 :- queen(I,J), queen(II,JJ), (I,J)!=(II,JJ), |I-II|=|J-JJ|.
```

The first rule of *8Queens* (Line 3), viewed as a one-rule program, has a unique stable model S, which satisfies the following condition:

$$\text{A ground atom of the form } row(i) \text{ belongs to } S \text{ iff } i \in \{1,\dots,8\}. \tag{1}$$

Condition (1) holds also if S is the stable model of the first two rules of this program. And it holds if S is any stable model of the first three rules, and so on, for all 6 "prefixes" (initial segments) of

[1] A concise introduction to ASP can be found in Chapter 1 of that book. Examples of programs in this paper are written in the input language of the grounder GRINGO, Version 5.

the program. This is what we mean by achievement: once the programmer declares that a property "has been achieved," he is committed to maintaining this property of stable models until the program is completed.

After writing the second rule (Line 4), the programmer can claim that something else has been achieved:

$$\text{A ground atom of the form } col(j) \text{ belongs to } S \text{ iff } j \in \{1,\ldots,8\}. \tag{2}$$

This condition holds if S is a stable model of any prefix of the program that includes the first two rules.

Additional properties achieved by adding the third rule can be expressed as follows:

$$\begin{array}{l}\text{Set } S \text{ contains exactly 8 ground atoms of the form } queen(i,j).\\ \text{For each of these atoms, } i,j \in \{1,\ldots,8\}.\end{array} \tag{3}$$

If a program is written in this manner then every achievement documented in the process of writing it describes a property shared by all stable models of the entire program. In some cases this list of achievements can serve as the skeleton of a proof of its correctness, in the spirit of Edsger Dijkstra's advice:

> ... one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand [Dijkstra, 1972].

Recording important achievements in the process of writing an ASP program may be similar to recording important loop invariants in procedural programming: it does not ensure the correctness of the program but helps the programmer move toward the goal of proving correctness.

A preliminary report on this project was presented at the 2016 Workshop on Answer Set Programming and Other Computing Paradigms. This is a corrected version of the paper that appeared in *Theory and Practice of Logic Programming*.

2 Programs, Prefixes, and Achievements

In this paper, by an (ordered) program we understand a list of rules $R_1,\ldots,R_n$ ($n \geq 1$) in the input language of an answer set solver, such as CLINGO [Gebser *et al.*, 2015b] or DLV [Eiter *et al.*, 1998]. The order of rules is supposed to reflect the order in which the programmer writes them in the process of creating the program. It does not affect the semantics of the program, but it is essential for understanding the process of programming.

We restrict attention to programs without classical negation. (This limitation is discussed in the conclusion.) Stable models of a program without classical negation are sets of ground atoms that contain no arithmetic operations, intervals, or pools [Gebser *et al.*, 2015b, Sections 3.1.7, 3.1.9, 3.1.10]. Such ground atoms will be called *precomputed*.[2] An *interpretation* is a set of precomputed atoms.

The *k-th prefix* of a program $R_1,\ldots,R_n$, where $1 \leq k \leq n$, is the program $R_1,\ldots,R_k$. We will express that a program Γ is a prefix of a program Π by writing $\Gamma \leq \Pi$. The relation $\leq$ is a total order on the set of prefixes of a program.

An *achievement* of a prefix Γ of Π is a property of sets of interpretations that holds for all stable models of all programs Δ such that $\Gamma \leq \Delta \leq \Pi$. For example, (1) is an achievement of the first prefix

[2] This terminology follows Gebser et al. [2015a], where "precomputed terms" are defined. Calimeri et al. [2012] talk about elements of the "Herbrand universe" of a program in the same sense.

of program *8Queens*; (2) is an achievement of its second prefix; and (3) is an achievement of its third prefix. Conditions (1) and (2), and the conjunction of conditions (1)–(3), are achievements of the third prefix as well. Any condition that holds for all sets of interpretations is trivially an achievement of any prefix of any program.

The following three conditions are achievements of the last three prefixes of *8Queens*:

Each column of the 8×8 chessboard includes at most one square (i, j) such that the atom $queen(i, j)$ belongs to S. (4)

Each row of the 8×8 chessboard includes at most one square (i, j) such that the atom $queen(i, j)$ belongs to S. (5)

Each diagonal of the 8×8 chessboard includes at most one square (i, j) such that the atom $queen(i, j)$ belongs to S. (6)

Thus every stable model S of *8Queens* satisfies all conditions (1)–(6).

3 Programs with Input

In some programs, constants are used as placeholders for values provided by the user [Gebser *et al.*, 2015b, Section 3.1.15]. For example, the constant n is used as a placeholder for an arbitrary positive integer in the following more general version of *8Queens*:

```
% Program NQueens

row(1..n).
col(1..n).
n { queen(I,J) : col(I), row(J) } n.
:- queen(I,J), queen(I,JJ), J!=JJ.
:- queen(I,J), queen(II,J), I!=II.
:- queen(I,J), queen(II,JJ), (I,J)!=(II,JJ), |I-II|=|J-JJ|.
```

The value of a placeholder is one kind of input that an answer set solver may expect in addition to the rules of the program. A definition of an "extensional predicate" occurring in the bodies of rules is another kind. Consider, for example, the following encoding of Hamiltonian cycles, adapted from [Gebser *et al.*, 2012, Section 3.3]:

```
% Program Hamiltonian

1 {in(X,Y) : edge(X,Y) } 1 :- vertex(X).
1 {in(X,Y) : edge(X,Y) } 1 :- vertex(Y).
reached(X) :- in(v0,X).
reached(Y) :- reached(X), in(X,Y).
:- not reached(X), vertex(X).
```

It needs to be supplemented by definitions of the predicate $vertex/1$, representing the set of vertices of a finite digraph G; of the predicate $edge/2$, representing the set of edges of G; and of the placeholder $v0$, which is a vertex of G.

In general, we understand an *input* as a function $\mathbf{i}$ defined on a finite set consisting of predicate symbols and symbolic constants such that

- if p/m is a predicate symbol in the domain of $\mathbf{i}$ then $\mathbf{i}(p/m)$ is a finite set of m-tuples of precomputed terms;
- if c is a symbolic constant in the domain of $\mathbf{i}$ then $\mathbf{i}(c)$ is a precomputed term.[3]

The result of *enriching* a program Π by an input $\mathbf{i}$, denoted by $\Pi \diamond \mathbf{i}$, is the program consisting of

- the facts $p(\mathbf{t})$ for all predicate symbols p/m in the domain of $\mathbf{i}$ and all tuples $\mathbf{t}$ in $\mathbf{i}(p/m)$, followed by
- the rules obtained from the rules of Π by substituting the terms $\mathbf{i}(c)$ for all occurrences of symbolic constants c in the domain of $\mathbf{i}$.

The stable models of $\Pi \diamond \mathbf{i}$ will be called the *stable models of* Π *for input* $\mathbf{i}$.

For example, if $\mathbf{i}$ is the input that maps n to 8 then *NQueens* $\diamond\, \mathbf{i}$ is the program *8Queens*. If $\mathbf{i}$ is the input that maps $vertex/1$ to $\{a,b\}$, $edge/2$ to $\{(a,b),(b,a)\}$, and *v0* to a, then *Hamiltonian* $\diamond\, \mathbf{i}$ is the program consisting of the facts

```
vertex(a).  vertex(b).  edge(a,b).  edge(b,a).
```

followed by the rules of *Hamiltonian* with *v0* replaced by a.

We will now extend the definition of an achievement to programs for which some inputs are designated as "valid." For example, we can say that an input $\mathbf{i}$ is considered valid for the program *NQueens* if its domain includes only one object—the symbolic constant n—and if $\mathbf{i}(n)$ is a positive integer. An input $\mathbf{i}$ is considered valid for the program *Hamiltonian* if (a) its domain consists of three objects—the predicate symbols $vertex/1$ and $edge/2$ and the symbolic constant *v0*; (b) $\mathbf{i}(edge/2)$ is the set of edges of a digraph with the set of vertices $\mathbf{i}(vertex/1)$; and (c) $\mathbf{i}(v0)$ is a vertex of that graph.

In this setting, an achievement is a relation between valid inputs and sets of interpretations. Such a relation A is an *achievement* of a prefix Γ of Π if, for every program Δ such that $\Gamma \le \Delta \le \Pi$, $A(\mathbf{i},S)$ holds for every valid input $\mathbf{i}$ and every stable model S of Δ for input $\mathbf{i}$. In particular, if A is an achievement of the entire program Π then $A(\mathbf{i},S)$ holds for every valid input $\mathbf{i}$ and every stable model S of Π for that input.

For example, the sentence

$$\text{A ground atom of the form } row(i) \text{ belongs to } S \text{ iff } i \in \{1,\dots,\mathbf{i}(n)\} \tag{7}$$

expresses a relation between $\mathbf{i}$ and S that is an achievement of the first prefix of *NQueens*. It is obtained from condition (1) by replacing 8 with $\mathbf{i}(n)$, and achievements of the other prefixes of that program can be obtained in a similar way from conditions (2)–(6).

The following two conditions are achievements of the first two prefixes of *Hamiltonian*:

> Every pair (x,y) such that the atom $in(x,y)$ belongs to S is an edge of the digraph G with the vertices $\mathbf{i}(vertex/1)$ and edges $\mathbf{i}(edge/2)$; for every vertex x of G there is a unique y such that the atom $in(x,y)$ belongs to S. (8)

[3] Programs with input in the sense of this definition are similar to lp-functions in the sense of Gelfond and Przymusinska [1996]. The description of an lp-function specifies not only its input, but also its output; on the other hand, the input of an lp-function includes predicates only, not placeholders.

$$\text{For every vertex } y \text{ of } G \text{ there is a unique } x \text{ such that the atom } in(x,y) \text{ belongs to } S. \tag{9}$$

Nothing interesting has been achieved by adding the third rule, but the following condition is an achievement of the fourth prefix of the program:

> The set of symbols x such that the atom $reached(x)$ belongs to S consists of the vertices x of G for which there exists a walk $v_0,\dots,v_n$ such that $n \geq 1$, $v_0 = v0$, $v_n = x$, and every atom of the form $in(v_i, v_{i+1})$ belongs to S. (10)

Finally, here is an achievement of the entire program:

$$\text{For every vertex } x \text{ of } G, \text{ the atom } reached(x) \text{ belongs to } S. \tag{11}$$

4 Records of Achievement

A *record of achievement* for a program Π is described by assigning an achievement A_Γ to each Γ in the record's *domain*, which is a set of prefixes of Π that includes the entire program Π. We will represent a record of achievement by the listing of the program with comments describing the achievements A_Γ placed vafter all prefixes Γ in the record's domain.

For example, here is the program *NQueens* with a record of achievement:

```
% Program NQueens, with a record of achievement

% input: positive integer n (the size of the board).

% A square on the board is represented as a pair, column
% number and row number, both from the set {1,..,n}.

row(1..n).
% achieved: row/1 = {1,...,n}.

col(1..n).
% achieved: col/1 = {1,...,n}.

n { queen(I,J) : col(I), row(J) } n.
% achieved: Set queen/2 consists of n squares.

:- queen(I,J), queen(I,JJ), J!=JJ.
% achieved: Each column includes at most one square from queen/2.

:- queen(I,J), queen(II,J), I!=II.
% achieved: Each row includes at most one square from queen/2.

:- queen(I,J), queen(II,JJ), (I,J)!=(II,JJ), |I-II|=|J-JJ|.
% achieved: Each diagonal includes at most one square from
%           queen/2.
```

The domain of this record achievement is the set of all prefixes of the program. The comment in Line 3 shows which inputs for the program are considered valid. The comment in Line 9 is a concise reformulation of condition (7). It uses $row/1$ as shorthand for "the set of precomputed terms i such that the atom $row(i)$ belongs to S." In the other comments, $col/1$ and $queen/2$ are understood in a similar way.

Program *Hamiltonian* with a record of achievement below uses another useful convention: in Lines 11 and 12, we understand X and Y as metavariables for precomputed terms. The comment in those lines is a reformulation of condition (9).

```
1  % Program Hamiltonian, with a record of achievement
2
3  % input: the set vertex/1 of vertices of a finite digraph G;
4  %        the set edge/2 of edges of G; a vertex v0 of G.
5
6  1 {in(X,Y) : edge(X,Y) } 1 :- vertex(X).
7  % achieved: Set in/2 is a subset of edge/2; for every vertex
8  %           X of G there is a unique Y such that in(X,Y).
9
10 1 {in(X,Y) : edge(X,Y) } 1 :- vertex(Y).
11 % achieved: For every vertex Y of G there is a unique X
12 %           such that in(X,Y).
13
14 reached(X) :- in(v0,X).
15 reached(Y) :- reached(X), in(X,Y).
16 % achieved: Set reached/1 consists of the vertices that are
17 %           reachable from v0 by a path of non-zero length
18 %           in the subgraph of G with the set of edges in/2.
19
20 :- not reached(X), vertex(X).
21 % achieved: reached/1 = vertex/1.
```

5 Completeness

The record of achievement for program *NQueens* in Section 4 shows that every stable model S of that program for the input $n = 8$ satisfies conditions (1)–(6). The converse is not true: some sets of precomputed atoms satisfying all these conditions are not stable models. This is clear from the fact that these conditions say nothing about precomputed atoms formed using predicate symbols other than

$$row/1,\ col/1,\ queen/2. \tag{12}$$

Adding such "irrelevant" atoms to a stable model of the program would not invalidate properties (1)–(6). It is true, however, that if every atom in S contains one of the symbols (12) then S has all these properties if *and only if* it is a stable model for $n = 8$.

This observation leads us to the following definitions. The *vocabulary* of a program Π is the set of all precomputed atoms $p(t_1, \ldots, t_n)$ such that the predicate symbol p/n occurs in Π. A record of achievement $\Gamma \mapsto A_\Gamma$ for Π is *complete* if, for every valid input $\mathbf{i}$, each subset S of the vocabulary of $\Pi \diamond \mathbf{i}$ that satisfies conditions $A_\Gamma(\mathbf{i}, S)$ for all prefixes Γ in the record's domain is a stable model of Π for input $\mathbf{i}$. The converse—every stable model S of Π for input $\mathbf{i}$ is a subset of the vocabulary satisfying these conditions—is true for any record of achievement. Thus achievements in a complete record provide a complete characterization of the class of stable models of the program.

For example, the record of achievement for *NQueens* in Section 4 is complete. This property can be lost if we make the achievements in that record weaker. For instance, if we replace "consists of n squares" in Line 15 by "consists of at most n squares" then we will not eliminate the possibility that $queen/2$ is empty.

The completeness property holds not only for the entire record of achievement for *NQueens* above, but also for its initial segments. For instance, a set S of precomputed atoms formed using $row/1$ satisfies (1) only if it is a stable model of the first rule of the program for $n = 8$. A set of precomputed atoms formed using $row/1$, $col/1$ satisfies (1) and (2) only if it is a stable model of the first two rules, and so forth.

The record of achievement for *Hamiltonian* in Section 4 is complete also, and so are its initial segments. For example, a set S of precomputed atoms formed using the predicate symbols $vertex/1$, $edge/2$, and $in/2$ satisfies condition (8) if and only if it is a stable model of the first rule of the program for input $\mathbf{i}$.

6 Achievement-Based Answer Set Programming

Both records of achievement given as examples in Section 4 are not only complete but also detailed, in the sense that they include achievements for almost all prefixes of the programs. The only rule in these programs that is not followed by an achievement comment is the first rule in the recursive definition of *reached*. The role of that rule cannot be properly explained unless we treat it as part of the definition.

Developing an ASP program along with a complete and detailed record of achievement can be called "achievement-based" answer set programming. This strategy allows us to set out our understanding of the design of the program by describing the roles of individual rules, or small groups of rules, in a mathematically precise way.

One of the advantages of this approach is that comments explaining what is achieved by a group of rules at the beginning of a program help us start testing and debugging it at an early stage, when only a part of the program has been written. To this end, we can run an answer set solver to find stable models of the prefix that has has been already written and check that they satisfy the conditions in the available "achieved" comments. A mismatch would indicate that there is a bug in the rules of the program written so far, or perhaps that the programmer's intentions have not been properly documented in the recorded achievements.

A complete record of achievement is particularly valuable when it is closely related to the program's specification, because from the completeness of such a record we may be able to conclude that the program is correct. For instance, from the completeness of the record of achievement of *NQueens* in Section 4 we can conclude that the stable models of that program are in a one-to-one correspondence with solutions to the n queens problem.

To further illustrate the idea of achievement-based ASP, we present below three "real life" ASP programs accompanied by complete, detailed records of achievement. The first of them, program *SCA* [Brain *et al.*, 2012, Figure 1], generates sequence covering arrays[4] of strength 3. Our version of *SCA* is slightly different from the original program: the constraint in Line 19 here replaces the pair of rules

[4] A sequence covering array of strength t is an array of permutations of symbols such that every ordering of any t symbols appears as a subsequence of at least one row.

```
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).
:- hb(N,X,X).
```

The reason why we chose to make this change is that the first of the two rules above may temporarily destroy the irreflexivity of the relation of hb_N that was true at the previous step; that property is restored by the second rule. That is not in the spirit of the achievement-based approach, which emphasizes the gradual accumulation of properties that we would like to see in the complete program.

```
% Program SCA

% input: the number s of symbols 1,...,s; the number n of
%        rows 1,...,n.

sym(1..s).
% achieved: sym/1 = {1,...,s}.

row(1..n).
% achieved: row/1 = {1,...,n}.

1 {hb(N,X,Y); hb(N,Y,X)} 1 :- row(N), sym(X), sym(Y), X!=Y.
% For every row N, let hb_N be the binary relation on sym/1
% defined by the condition: X hb_N Y iff hb(N,X,Y).
% achieved: each relation hb_N is irreflexive; each pair of
%           distinct symbols satisfies either X hb_N Y
%           or Y hb_N X.

:- hb(N,X,Y), hb(N,Y,Z), not hb(N,X,Z).
% achieved: each relation hb_N is transitive.

covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
% For every row N and every symbol X, by M_{N,X} we denote
% the symbol that is the X-th smallest w.r.t. hb_N.
% achieved: for any symbols X, Y, Z, covered(X,Y,Z) iff,
%           for some row N, (X,Y,Z) is a subsequence of
%           (M_{N,1},...,M_{N,s}).

:- not covered(X,Y,Z), sym(X), sym(Y), sym(Z),
   X!=Y, Y!=Z, X!=Z.
% achieved: covered(X,Y,Z) for any pairwise distinct symbols
%           X, Y, Z.
```

The other two examples are program *Borda*, adapted from [Charwat and Pfandler, 2015, Encoding 1], which encodes the Borda rule for determining the winner in an election with several candidates,[5]

[5] Each voter ranks the list of candidates in order of preference. The candidate ranked last gets zero points; next to last gets one point, and so on. The candidate with the most points is the winner.

```
% Program Borda

% input: the number m of candidates 1,...,m in an election
%        E; the set p/3 of triples (P,Pos,C) such that, for
%        a fixed ordering pr_1,...,pr_l of the distinct
%        preference relations in the profile of E, candidate
%        C is at position Pos in relation pr_P; the set
%        votecount/2 of pairs (P,VC) such that relation pr_P
%        occurs VC times in the profile of E.

candidate(1..m).
% achieved: candidate/1 = {1,...,m}.

posScore(P,C,X*VC) :- p(P,Pos,C), X=m-Pos, votecount(P,VC).
% achieved: posScore(P,C,S) iff the voters who chose
%           relation pr_P in election E contributed S points
%           to candidate C under the Borda rule.

score(C,N) :- candidate(C), N=#sum{S,P:posScore(P,C,S)}.
% achieved: score(C,N) iff candidate C earned N points in
%           election E under the Borda rule.

winner(C) :- score(C,M), M=#max{S:score(_,S)}.
% achieved: winner(C) iff the number of points earned by
%           candidate C in election E is maximal among all
%           candidates.
```

and program *OBT*, adapted from [Brooks *et al.*, 2007, Section 1], which encodes ordered binary trees.[6]

```
% Program OBT

% input: positive integer k.

leaf(0..k).
% achieved: leaf/1 = {0,...,k}.

vertex(0..2*k).
% achieved: vertex/1 = {0,...,2k}.

internal(X) :- vertex(X), not leaf(X).
% achieved: internal/1 = {k+1,...,2k}.

2 {edge(X,Y) : vertex(Y), X>Y} 2 :- internal(X).
% Let G be the digraph with the vertices vertex/1 and the
% edges edge/2.
% achieved: for every edge (X,Y) of G, X>Y; the out-degree
%           of a vertex X in G is 2 if internal(X), and 0
%           if leaf(X).

reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z), reachable(Z,Y).
% achieved: reachable(X,Y) iff Y is reachable from X in G
%           by a path of non-zero length.

:- vertex(X), X!=2*k, not reachable(2*k,X).
% achieved: every vertex of G other than 2k is reachable
%           from 2k by a path of non-zero length.

:- reachable(X,X), vertex(X).
% achieved: G is acyclic.

max_child(X,Y) :- edge(X,Y), edge(X,Y1), Y > Y1.
% achieved: max_child(X,Y) iff Y is the largest child of X
%           in G.

Y<Y1 :- max_child(X,Y), max_child(X1,Y1), Y>Y1, X<X1.
% achieved: for any vertices X, X1 of G such that X<X1, the
%           largest child of X is smaller than the largest
%           child of X1.
```

[6] An ordered binary tree is a rooted binary tree with the leaves $0,\dots,k$ and internal vertices $k+1,\dots,2k$ such that (i) every internal vertex is greater than its children, and (ii) for any two internal vertices x and x_1, $x > x_1$ iff the maximum of the children of x is grater than the maximum of the children of x_1.

7 Achievements in Teaching

The achievement-based approach was emphasized in a class on answer set programming taught recently to a group of over 50 undergraduates at the University of Texas at Austin. The idea of an achievement was explained more informally than in this paper, but many examples were given. In most solutions to programming assignments submitted for grading, students attempted to imitate the instructor's use of "input" and "achieved" comments, even though they were not instructed to do that. The degree of their success depended, of course, on their previous exposure to logic and mathematics. When ASP programs written by students were discussed in class, the instructor emphasized the difference between the correctness of the program on the one hand, and the clarity and correctness of "input" and "achieved" comments on the other.

Comments of these kinds can be used in exercises and test problems. In one case, students were shown an "incomplete listing" of a graph coloring program:

```
1  % Color the vertices of a graph so that no two adjacent
2  % vertices share the same color.
3
4  % input: set vertex/1 of vertices of a graph G;
5  %        set edge/2 of edges of G; set color/1 of colors.
6
7  1 {color(X,C) : color(C)} 1 :- vertex(X).
8  % achieved: for every vertex X there is a unique color C
9  %           such that color(X,C).
10
11 ---------------------------------------------------------
12 % achieved: no two adjacent vertices share the same color.
13
14 #show color/2.
```

The question was, "What rule would you place in Line 11?" On another occasion, students were asked to write a one-rule program for which the following comments would be appropriate:

```
% Calculate the number of classes taught today on each of
% the seven floors of the computer science building.

% input: set where/2 of all pairs (C,I) such that class C
%        is taught on the I-th floor.


---------------------------------------------------------
% achieved: howmany(I,N) iff the number of classes taught
%           on the I-th floor is N.

#show howmany/2.
```

8 Conclusion

In achievement-based ASP, we start writing a program by describing its inputs. Then, after every rule or small group of rules, we include a comment describing what has been achieved. Collectively these comments represent a complete, detailed record of achievement.

As we are adding rules to an emerging ASP program, we deal at every step with a single executable piece of code, unlike the non-executable pseudo-code formed in the process of stepwise refinement of a procedural program, and unlike a collection of executable subroutines formed in the process of bottom-up design. In the process, we think of prefixes of the emerging program as if they were complete programs. We describe their stable models in a way that relates them to the stable models of the final product.

The programs discussed in this paper do not use classical negation [Gelfond and Lifschitz, 1990]. In the presence of classical negation, answer sets consist of "precomputed classical literals" —precomputed atoms and classical negations of such atoms. Extending the definition of a complete record of achievement to such programs is straightforward. On the other hand, many programs with classical negation contain defaults [Gelfond and Kahl, 2014, Chap. 5], such as the closed world assumption and the commonsense law of inertia, and the achievement-based approach may be not so useful in application to programs containing defaults. A default does not "achieve" anything in the technical sense of Section 2.

When ASP is used for representing dynamic domains, a very different methodology can be recommended: first describe the domain in an action description language, and then translate its causal laws into answer set programming [Gelfond and Lifschitz, 1993], [Lifschitz and Turner, 1999], [Gelfond and Kahl, 2014].

According to Gebser et al. [2012],

> [t]he basic approach to writing encodings in ASP follows a *generate-and-test* methodology, also referred to as *guess-and-check*... A "generating" part is meant to non-deterministically provide solution candidates, while a "testing" part eliminates candidates violating some requirements... Both parts are usually amended by "defining" parts providing auxiliary concepts.

Most programs discussed in this paper are designed in accordance with this basic approach.[7] The advice to keep track of what has been achieved as you are adding rules to your program differs from the "generate-and-test" advice in that it refers to mathematical properties of stable models, and not to programmer's intentions.

Acknowledgements

Thanks to Michael Gelfond, Amelia Harrison, Yuliya Lierler, Julian Michael, Liangkun Zhao, and the anonymous referees for comments on earlier versions of this paper. Conversations and exchanges of email messages with Mark Denecker, Esra Erdem, Martin Gebser, Roland Kaminski, Johannes Oetsch, Dhananjay Raju, and Mirek Truszczynski helped the author develop a better understanding of the methodology of answer set programming. This research was partially supported by the National Science Foundation under Grant IIS-1422455.

[7] Program *Borda* is an exception—it has no generating part and no testing part. Also, it is not clear whether the designers of *Hamiltonian* intended the second rule for the generating part or for the testing part. (The second rule is syntactically similar to the first, which is definitely a generate rule. On the other hand, adding the second rule does not really generate new solution candidates; it eliminates some of the candidates generated earlier.)

References

[Brain *et al.*, 2012] Martin Brain, Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jörg Pührer, Hans Tompits, and Cemal Yilmaz. Event-sequence testing using answer-set programming. *International Journal on Advances in Software*, 5:237–251, 2012.

[Brooks *et al.*, 2007] Daniel R. Brooks, Esra Erdem, Selim T. Erdoğan, James W. Minett, and Donald Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 39:471–511, 2007.

[Calimeri *et al.*, 2012] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf`, 2012.

[Charwat and Pfandler, 2015] Günther Charwat and Andreas Pfandler. Democratix: A declarative approach to winner determination. In *Proceedings of the 4th International Conference on Algorithmic Decision Theory (ADT)*, 2015.

[Dijkstra, 1972] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15:859–866, 1972.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system DLV: Progress report, comparisons and benchmarks. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 406–417, 1998.

[Gebser *et al.*, 2012] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[Gebser *et al.*, 2015a] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. Abstract Gringo. *Theory and Practice of Logic Programming*, 15:449–463, 2015.

[Gebser *et al.*, 2015b] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, and Sven Thiele. Potassco User Guide, version 2.0. Available at `http://potassco.sourceforge.net`, 2015.

[Gelfond and Kahl, 2014] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, 2014.

[Gelfond and Lifschitz, 1990] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In David Warren and Peter Szeredi, editors, *Proceedings of International Conference on Logic Programming (ICLP)*, pages 579–597, 1990.

[Gelfond and Lifschitz, 1993] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[Gelfond and Przymusinska, 1996] Michael Gelfond and Halina Przymusinska. Towards a theory of elaboration tolerance: Logic programming approach. *International Journal of Software Engineering and Knowledge Engineering*, 6(1):89–112, 1996.

[Lifschitz and Turner, 1999] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Proceedings of International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR)*, pages 92–106, 1999.

[Marek and Truszczynski, 1999] Victor Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[Niemelä, 1999] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.