Appendix A Saturation-based meta encoding

The saturation-based meta encoding in Listing 47 relies on a partition of the atoms of the input program induced by the strongly connect components of its positive dependency graph. Each atom and each loop of the program is contained in some part. The idea is to mimic the consecutive application of the immediate consequence operator to each component of the partition.

In a nutshell, the encoding in Listing 47 combines the following parts (Gebser et al. 2011):

1. guessing an interpretation (in Lines 14 to 33),
2. deriving the unsatisfiability-indicating atom `bot` if the interpretation is not a supported model (where each true atom occurs positively in the head of some rule whose body holds; cf. Lines 35 and 36),
3. deriving `bot` if the true atoms of some non-trivial strongly connected component are not acyclicly derivable (checked via determining the complement of a fixpoint of the immediate consequence operator; cf. Lines 38 to 63), and
4. saturating interpretations that do not correspond to stable models by deriving all truth assignments (for atoms) from `bot` (in Lines 65 and 66).

As an example, consider the simple logic program `a.lp`:

```
1 { a(1..2) }.
```

Computing its stable models with the meta encoding in Listing 47 (along with the auxiliary `#show` statements from Listing 48) yields the three expected models:

```
UNIX> clingo --output=reify --reify-sccs a.lp | \
      clingo - metaD.lp show.lp  0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
a(2)
Answer: 2
a(1)
Answer: 3
a(1) a(2)
SATISFIABLE
```

Now, the addition of an empty integrity constraint, namely "`:-.`", makes the program unsatisfiable. This is reflected by a single answer set containing all atoms of the program. This should not to be confused with the third model obtained above:

```
UNIX> clingo --output=reify --reify-sccs a.lp <(echo ":-.") | \
      clingo - metaD.lp show.lp  0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
a(1) a(2)
SATISFIABLE
```

In fact, additional show statements would reveal that the actual stable model also contains the artificial atom `bot` from which all atoms occurring in the original program are derivable (cf. Lines 65 and 66 in Listing 47). In other words, this special atom expresses the non-existence of stable models, and by saturating the model with all atoms it can only exist if no true stable models exist. This is because the semantics of disjunctive logic programs

```
sum(B,G,T) :- rule(_,sum(B,G)), T = #sum { W,L: weighted_literal_tuple(B,L,W) }.

supp(A,B) :- rule(     choice(H),B), atom_tuple(H,A).
supp(A,B) :- rule(disjunction(H),B), atom_tuple(H,A).

supp(A) :- supp(A,_).

atom(|L|) :- weighted_literal_tuple(_,L,_).
atom(|L|) :- literal_tuple(_,L).
atom( A ) :- atom_tuple(_,A).

fact(A) :- rule(disjunction(H),normal(B)), atom_tuple(H,A), not literal_tuple(B,_).

true(atom(A))                 :- fact(A).
true(atom(A)); fail(atom(A)) :- supp(A), not fact(A).
               fail(atom(A)) :- atom(A), not supp(A).

true(normal(B)) :- literal_tuple(B),
  true(atom(L)): literal_tuple(B, L), L>0;
  fail(atom(L)): literal_tuple(B,-L), L>0.
fail(normal(B)) :- literal_tuple(B, L), fail(atom(L)), L>0.
fail(normal(B)) :- literal_tuple(B,-L), true(atom(L)), L>0.

true(sum(B,G)) :- sum(B,G,T),
  #sum {
    W,L: true(atom(L)), weighted_literal_tuple(B, L,W), L>0;
    W,L: fail(atom(L)), weighted_literal_tuple(B,-L,W), L>0
  } >= G.
fail(sum(B,G)) :- sum(B,G,T),
  #sum {
    W,L: fail(atom(L)), weighted_literal_tuple(B, L,W), L>0;
    W,L: true(atom(L)), weighted_literal_tuple(B,-L,W), L>0
  } >= T-G+1.

bot :- rule(disjunction(H),B), true(B), fail(atom(A)): atom_tuple(H,A).
bot :- true(atom(A)), fail(B): supp(A,B).

internal(C,normal(B)) :- scc(C,A), supp(A,normal(B)), scc(C,A'),
                         literal_tuple(B,A').
internal(C,sum(B,G))  :- scc(C,A), supp(A,sum(B,G)),  scc(C,A'),
                         weighted_literal_tuple(B,A',W).

external(C,normal(B)) :- scc(C,A), supp(A,normal(B)), not internal(C,normal(B)).
external(C,sum(B,G))  :- scc(C,A), supp(A,sum(B,G)),  not internal(C,sum(B,G)).

steps(C,Z-1) :- scc(C,_), Z = { scc(C,A): not fact(A) }.

wait(C,atom(A),0)   :- scc(C,A), fail(B): external(C,B), supp(A,B).
wait(C,normal(B),I) :- internal(C,normal(B)), steps(C,Z), I=0..Z-1,
                       fail(normal(B)).
wait(C,normal(B),I) :- internal(C,normal(B)), steps(C,Z), I<Z,
                       literal_tuple(B,A), wait(C,atom(A),I).

wait(C,sum(B,G),I)  :- internal(C,sum(B,G)), steps(C,Z), I=0..Z-1, sum(B,G,T),
  #sum {
    W,L:   fail(atom(L)),   weighted_literal_tuple(B, L,W), L>0, not scc(C,L);
    W,L: wait(C,atom(L),I), weighted_literal_tuple(B, L,W), L>0,     scc(C,L);
    W,L:   true(atom(L)),   weighted_literal_tuple(B,-L,W), L>0
  } >= T-G+1.
wait(C,atom(A),I)   :- wait(C,atom(A),0), steps(C,Z), I=1..Z,
                       wait(C,B,I-1): supp(A,B), internal(C,B).

bot :- scc(C,A), true(atom(A)), wait(C,atom(A),Z), steps(C,Z).

true(atom(A)) :- supp(A), not fact(A), bot.
fail(atom(A)) :- supp(A), not fact(A), bot.
```

Listing 47. A disjunctive meta encoding implementing saturation (`metaD.lp`)

```
#show.
#show X: output(X,B),     literal_tuple(B,A), true(atom(A)).
#show X: output(X,B), not literal_tuple(B,_).
```

Listing 48. Auxiliary `#show` statements for Listing 47 (`show.lp`)

is based on subset minimization. Saturation makes sure that `bot` is derived only if it is inevitable, that is, if it is impossible to construct any other models.[26]

Appendix B Intermediate language

To accommodate the rich input language, a general grounder-solver interface is needed. Although this could be left internal to *clingo*, it is good practice in ASP and neighboring fields to explicate such interfaces via an intermediate language. This also allows for using alternative downstream solvers or transformations.

Unlike the block-oriented *smodels* format, the *aspif* format is line-based. Notably, it abolishes the need of using symbol tables in *smodels*' format (Syrjänen 2001) for passing along meta-expressions and allows *gringo* to output information as soon as it is grounded. An *aspif* file starts with a header, beginning with the keyword `asp` along with version information and optional tags, viz.

$$\texttt{asp}\textvisiblespace v_m \textvisiblespace v_n \textvisiblespace v_r \textvisiblespace t_1 \textvisiblespace \dots \textvisiblespace t_k$$

where v_m, v_n, v_r are non-negative integers representing the version in terms of *major*, *minor*, and *revision* numbers, and each t_i is a tag for $k \geq 0$. Currently, the only tag is `incremental`, meant to set up the underlying solver for multi-shot solving. An example header is given in the first lines of Listings 49 and 50 below. The rest of the file comprises one or more logic programs. Each logic program is a sequence of lines of *aspif* statements followed by a `0`, one statement or `0` per line, respectively. Positive and negative integers are used to represent positive or negative literals, respectively. Hence, `0` is not a valid literal.

Let us now briefly describe the format of *aspif* statements and illustrate them with the simple logic program in Listing 1 as well as the result of grounding a subset of Listing 33 only pertaining to difference constraints in Listing 50.

Rule statements have form

$$\texttt{1}\textvisiblespace H \textvisiblespace B$$

in which head H has form

$$h \textvisiblespace m \textvisiblespace a_1 \textvisiblespace \dots \textvisiblespace a_m$$

where $h \in \{\texttt{0}, \texttt{1}\}$ determines whether the head is a disjunction or a choice, $m \geq 0$ is the number of head elements, and each a_i is an atom.

Body B has one of two forms:

[26] In fact, without the two saturating rules in Lines 65 and 66, Listing 47 would produce a stable model for each interpretation of the original program. The ones without `bot` represent stable models, while the ones with `bot` are mere interpretations. By saturation, all these interpretations are mapped to the set of all atoms. Given that the latter is a superset of all conceivable stable models, it can only exist if no stable models exist.

```
1 asp 1 0 0
2 1 1 1 1 0 0
3 1 0 1 2 0 1 1
4 1 0 1 3 0 1 -1
5 4 1 a 1 1
6 4 1 b 1 2
7 4 1 c 1 3
8 0
```

Listing 49. Representing the logic program from Listing 1 in *aspif* format

- Normal bodies have form

  $$\texttt{0}\textvisiblespace n\textvisiblespace l_1\textvisiblespace\dots\textvisiblespace l_n$$

  where $n \geq 0$ is the length of the rule body, and each l_i is a literal.
- Weight bodies have form

  $$\texttt{1}\textvisiblespace l\textvisiblespace n\textvisiblespace l_1\textvisiblespace w_1\textvisiblespace\dots\textvisiblespace l_n\textvisiblespace w_n$$

  where l is a positive integer to denote the lower bound, $n \geq 0$ is the number of literals in the rule body, and each l_i and w_i are a literal and a positive integer.

All types of ASP rules are included in the above rule format. Heads are disjunctions or choices, including the special case of one-element disjunctions for representing normal rules. As in the *smodels* format, aggregate rules are restricted to one-element bodies, just that in *aspif* cardinality constraints are taken as special weight constraints. Otherwise, a body is simply a conjunction of literals.

The three rules in Listing 1 are represented by the statements in Lines 2–4 of Listing 49. For instance, the four occurrences of `1` in Line 2 capture a rule with a choice in the head, having one element, identified by `1`. The two remaining zeros capture a normal body with no element. For another example, Lines 2–7 of Listing 50 represent 6 of the facts in Listing 34, the four regular atoms in Lines 1–4 along two comprising theory atoms in Lines 11 and 12.

Minimize statements have form

$$\texttt{2}\textvisiblespace p\textvisiblespace n\textvisiblespace l_1\textvisiblespace w_1\textvisiblespace\dots\textvisiblespace l_n\textvisiblespace w_n$$

where p is an integer priority, $n \geq 0$ is the number of weighted literals, each l_i is a literal, and each w_i is an integer weight. Each of the above expressions gathers weighted literals sharing the same priority p from all `#minimize` directives and weak constraints in a logic program. As before, maximize statements are translated into minimize statements.

Projection statements result from `#project` directives and have form

$$\texttt{3}\textvisiblespace n\textvisiblespace a_1\textvisiblespace\dots\textvisiblespace a_n$$

where $n \geq 0$ is the number of atoms, and each a_i is an atom.

Output statements result from `#show` directives and have form

$$\texttt{4}\textvisiblespace m\textvisiblespace s\textvisiblespace n\textvisiblespace l_1\textvisiblespace\dots\textvisiblespace l_n$$

where $n \geq 0$ is the length of the condition, each l_i is a literal, and $m \geq 0$ is an integer indicating the length in bytes of string s (where s excludes byte '\0' and newline). The output statements in Lines 5–7 of Listing 49 print the symbolic representation of atom `a`, `b`, or `c`, whenever the corresponding atom is true. For instance, the string '`a`' is printed if atom '`1`' holds. Unlike this, the statements in Lines 8–11 of Listing 50 unconditionally print the symbolic representation of the atoms stemming from the four facts in Lines 1–4 of Listing 34.

External statements result from `#external` directives and have form

$$\mathtt{5}\sqcup a\sqcup v$$

where a is an atom, and $v \in \{0, 1, 2, 3\}$ indicates free, true, false, and release.

Assumption statements have form

$$\mathtt{6}\sqcup n\sqcup l_1\sqcup \dots \sqcup l_n$$

where $n \geq 0$ is the number of literals, and each l_i is a literal. Assumptions instruct a solver to compute stable models containing $l_1, \dots, l_n$. They are only valid for a single solver call.

Heuristic statements result from `#heuristic` directives and have form

$$\mathtt{7}\sqcup m\sqcup a\sqcup k\sqcup p\sqcup n\sqcup l_1\sqcup \dots \sqcup l_n$$

where $m \in \{0, \dots, 5\}$ stands for the (m+1)th heuristic modifier among `level`, `sign`, `factor`, `init`, `true`, and `false`, a is an atom, k is an integer, p is a non-negative integer priority, $n \geq 0$ is the number of literals in the condition, and the literals l_i are the condition under which the heuristic modification should be applied.

Edge statements result from `#edge` directives and have form

$$\mathtt{8}\sqcup u\sqcup v\sqcup n\sqcup l_1\sqcup \dots \sqcup l_n$$

where u and v are integers representing an edge from node u to node v, $n \geq 0$ is the length of the condition, and the literals l_i are the condition for the edge to be present.

Let us now turn to the theory-specific part of *aspif.* Once a theory expression is grounded, *gringo* outputs a serial representation of its syntax tree. To illustrate this, we give in Listing 50 the (sorted) result of grounding all lines of Listing 33 related to difference constraints, viz. Lines 1–20 and Line 24.

```
 1  asp 1 0 0
 2  1 0 1 1 0 0
 3  1 0 1 2 0 0
 4  1 0 1 3 0 0
 5  1 0 1 4 0 0
 6  1 0 1 5 0 0
 7  1 0 1 6 0 0
 8  4 7 task(1) 0
 9  4 7 task(2) 0
10  4 15 duration(1,200) 0
11  4 15 duration(2,400) 0
12  9 0 1 200
13  9 0 3 400
14  9 0 6 1
```

```
15  9 0 11 2
16  9 1 0 4 diff
17  9 1 2 2 <=
18  9 1 4 1 -
19  9 1 5 3 end
20  9 1 8 5 start
21  9 2 7 5 1 6
22  9 2 9 8 1 6
23  9 2 10 4 2 7 9
24  9 2 12 5 1 11
25  9 2 13 8 1 11
26  9 2 14 4 2 12 13
27  9 4 0 1 10 0
28  9 4 1 1 14 0
29  9 6 5 0 1 0 2 1
30  9 6 6 0 1 1 2 3
31  0
```

Listing 50. *aspif* format (excerpt of result)

Theory terms are represented using the following statements:

$$\texttt{9}_\sqcup\texttt{0}_\sqcup u_\sqcup w \tag{B1}$$

$$\texttt{9}_\sqcup\texttt{1}_\sqcup u_\sqcup n_\sqcup s \tag{B2}$$

$$\texttt{9}_\sqcup\texttt{2}_\sqcup u_\sqcup t_\sqcup n_\sqcup u_{1\sqcup} \dots {}_\sqcup u_n \tag{B3}$$

where $n \geq 0$ is a length, index u is a non-negative integer, integer w represents a numeric term, string s of length n represents a symbolic term (including functions) or an operator, integer t is either `-1`, `-2`, or `-3` for tuple terms in parentheses, braces, or brackets, respectively, or an index of a symbolic term or operator, and each u_i is an integer for a theory term. Statements (B1), (B2), and (B3) capture numeric terms, symbolic terms, as well as compound terms (tuples, sets, lists, and terms over theory operators).

Fifteen theory terms are given in Lines 12–26 of Listing 50. Each of them is identified by a unique index in the third spot of each statement. While Lines 12–20 stand for primitive entities of type (B1) or (B2), the ones beginning with '$\texttt{9}_\sqcup\texttt{2}$' represent compound terms. For instance, Lines 21 and 22 represent `end(1)` or `start(1)`, respectively, and Line 23 corresponds to `end(1)-start(1)`.

Theory atoms are represented using the following statements:

$$\texttt{9}_\sqcup\texttt{4}_\sqcup v_\sqcup n_\sqcup u_{1\sqcup} \dots {}_\sqcup u_{n\sqcup} m_\sqcup l_{1\sqcup} \dots {}_\sqcup l_m \tag{B4}$$

$$\texttt{9}_\sqcup\texttt{5}_\sqcup a_\sqcup p_\sqcup n_\sqcup v_{1\sqcup} \dots {}_\sqcup v_n \tag{B5}$$

$$\texttt{9}_\sqcup\texttt{6}_\sqcup a_\sqcup p_\sqcup n_\sqcup v_{1\sqcup} \dots {}_\sqcup v_{n\sqcup} g_\sqcup u_1 \tag{B6}$$

where $n \geq 0$ and $m \geq 0$ are lengths, index v is a non-negative integer, a is an atom or `0` for directives, each u_i is an integer for a theory term, each l_i is an integer for a literal, integer p refers to a symbolic term, each v_i is an integer for a theory atom element, and integer g refers to a theory operator. Statement (B4) captures elements of theory atoms and directives, and statements (B5) and (B6) refer to the latter.

For instance, Line 27 captures the (single) theory element in '`{ end(1)-start(1) }`', and Line 29 represents the theory atom '`&diff { end(1)-start(1) } <= 200`'.

Comments have form

$$\texttt{10}\llcorner\!\lrcorner s$$

where s is a string not containing a newline.

The *aspif* format constitutes the default output of *gringo* 5. With *clasp* 3.2, ground logic programs can be read in both *smodels* and *aspif* format. The tool *lpconvert* can be used to convert between both formats (Potassco Team 2021f).