In: Proc. 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7), I. Niemelä and V. Lifschitz, editors, LNCS, © 2004 Springer.

Towards Automated Integration of Guess and Check Programs in Answer Set Programming *

Thomas Eiter[1] and Axel Polleres[2]

[1] Institut für Informationssysteme, TU Wien, A-1040 Wien, Austria
eiter@kr.tuwien.ac.at
[2] Institut für Informatik, Universität Innsbruck, A-6020 Innsbruck, Austria
axel.polleres@uibk.ac.at

Abstract. Many NP-complete problems can be encoded in the answer set semantics of logic programs in a very concise way, where the encoding reflects the typical "guess and check" nature of NP problems: The property is encoded in a way such that polynomial size certificates for it correspond to stable models of a program. However, the problem-solving capacity of full disjunctive logic programs (DLPs) is beyond NP at the second level of the polynomial hierarchy. While problems there also have a "guess and check" structure, an encoding in a DLP is often non-obvious, in particular if the "check" itself is co-NP-complete; usually, such problems are solved by interleaving separate "guess" and "check" programs, where the check is expressed by inconsistency of the check program. We present general transformations of head-cycle free (extended) logic programs into stratified disjunctive logic programs which enable one to integrate such "guess" and "check" programs automatically into a single disjunctive logic program. Our results complement recent results on meta-interpretation in ASP, and extend methods and techniques for a declarative "guess and check" problem solving paradigm through ASP.

1 Introduction

Answer set programming (ASP) [19, 7] is widely proposed as a useful tool for expressing properties in NP, where solutions and polynomial time proofs for such properties correspond to answer sets of normal logic programs, which cover by well-known complexity results the class NP. An example for such a property is whether a given graph has a legal 3-coloring, where any such coloring is itself a certificate for this property.

However, we also might encounter situations in which we want to express a problem which is complementary to some NP problem, and thus belongs to the class co-NP; it is widely believed that in general, not all such problems are in NP and hence not always a polynomial-size certificate checkable in polynomial time exists. One such problem is, e.g., the property that a graph is *not* 3-colorable. Such properties p can analogously be expressed by a normal logic program (equivalently, by a head-cycle free disjunctive logic program [1]) Π_p, where the property holds iff Π_p has no answer set at all.

Checks in co-NP typically occur as subproblems within more complex problems which have complexity higher than NP, for instance:

Quantified Boolean Formulas (QBFs): Evaluating a QBF, where we have to check, given a QBF of the form $\exists X \forall Y \Phi(X, Y)$, and an assignment σ to the variables X,

* The major part of this work has been conducted at TU Wien, supported by FWF (Austrian Science Funds) projects P14781 and Z29-N04 and European Commission grants FET-2001-37004 WASP and IST-2001-33570 INFOMIX.

whether $\forall Y \Phi(\sigma(X), Y)$ evaluates to true.

Strategic Companies: Checking whether a set of companies is strategic (cf. [11]).

Conformant Planning: Checking whether a given plan is conformant [8], provided executability of actions is polynomially decidable (cf. [4, 22]).

Further examples can be found in [6, 5]. In general, the corresponding logic program Π_p for this check can be easily formulated and the overall problem (evaluating the QBF, finding a strategic companies set resp. a conformant plan) solved in a 2-step approach:

1. Generate a candidate solution by means of a logic program Π_{guess}.
2. Check the solution by another logic program Π_{check} (=Π_p).

However, it is often not clear how to combine Π_{guess} and Π_{check} into a *single* program Π_{solve} which solves the overall problem. Simply taking the union $\Pi_{guess} \cup \Pi_{check}$ does not work, and rewriting is needed. Theoretical results [6] informally give strong evidence that for problems with Σ_2^P-complexity, it is required that Π_{check} (given as a normal logic program or a head-cycle free disjunctive logic program) is rewritten into a disjunctive logic program Π'_{check} such that the answer sets of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$ yield the solutions of the problem, where Π'_{check} emulates the inconsistency check for Π'_{check} as a minimal model check, which is co-NP-complete for disjunctive programs. This becomes even more complicated by the fact that Π'_{check} must not crucially rely on the use of negation, since it is essentially determined by the Π_{guess} part. These difficulties can make rewriting Π_{check} to Π'_{check} a formidable and challenging task.

In this paper, we present a generic method for rewriting Π_{check} automatically by using a meta-interpreter approach. In particular, we make the following contributions:

(1) We provide a polynomial-time transformation $tr(\Pi)$ from propositional head-cycle-free [1] (extended) disjunctive logic programs (HDLPs) Π to disjunctive logic programs (DLPs), such that the following conditions hold:

- **T1** Each answer set S' of $tr(\Pi)$ corresponds to an answer set S of Π, such that $S = \{l \mid \mathtt{inS}(l) \in S'\}$ for some predicate $\mathtt{inS}(\cdot)$.
- **T2** If the original program has no answer sets, then $tr(\Pi)$ has exactly one designated answer set Ω, which is easily recognizable.
- **T3** The transformation is of the form $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$, where $F(\Pi)$ is a factual representation of Π and Π_{meta} is a fixed *meta-interpreter*.
- **T4** $tr(\Pi)$ is *modular* (at the syntactic level), i.e., $tr(\Pi) = \bigcup_{r \in \Pi} tr(r)$. Moreover, it is a stratified DLP [20, 21] and uses negation only in its "deterministic" part.

We also describe optimizations and a transformation to positive DLPs, and show that in a precise sense, modular transformations to such programs do not exist.

(2) We show how to use $tr(\cdot)$ for integrating separate guess and check programs Π_{guess} and Π_{check}, respectively, into a single DLP Π_{solve} such that the answer sets of Π_{solve} yield the solutions of the overall problem.

(3) We demonstrate the method on the examples of QBFs and conformant planning [8] under fixed polynomial plan length (cf. [4, 22]), where our method proves to loosen some restrictions of previous encodings.

Our work enlarges the range of techniques for expressing problems using ASP, in a direction which to our knowledge has not been explored so far. It also complements recent results about meta-interpretation in ASP [16, 2, 3]. We fruitfully exploit the construction of $tr(\cdot)$ to further elucidate the natural guess and check programming paradigm for ASP, as discussed in [11] or in [14] (named "Generate/Define/Test" there), and we fill a gap by providing an automated construction for integrating guess and check programs. It is worth noticing that such an integration is non-trivial even for manual construction in general. Apart from being pure ASP solutions, integrated encodings may be straight subject to automated program optimization within ASP solvers, considering both the guess and check part as well as their interaction; this is not immediate for separate programs.

For space constraints, most proofs and longer encodings are omitted here. All proofs and further details (encodings, etc) are given in an extended version of this paper.[3]

2 Preliminaries

We assume that the reader is familiar with logic programming and answer set semantics (see [7, 19]) and only briefly recall the necessary concepts.

A *literal* is an atom $a(t_1,\ldots,t_n)$,or its negation $\neg a(t_1,\ldots,t_n)$, where "$\neg$" (alias, "–") is the strong negation symbol, in a function-free first-order language with at least one constant, which is customarily given by the programs considered. By $|a| = |\neg a| = a$ we denote the atom of a literal. Extended disjunctive logic programs (EDLPs; or simply programs) are finite sets Π of rules r

$$h_1 \;\texttt{v}\; \ldots \;\texttt{v}\; h_l \;\texttt{:-}\; b_1,\;\ldots,\;b_m,\;\texttt{not}\;b_{m+1},\;\ldots\;\texttt{not}\;b_n. \quad (1)$$

$l,m,n \geq 0$, where each h_i and b_j is a literal and $\texttt{not}$ is weak negation (negation as failure). By $H(r) = \{h_1,\ldots,h_l\}$, $B^+(r) = \{b_1,\ldots,b_m\}$, $B^-(r) = \{b_{m+1},\ldots,b_n\}$, and $B(r) = B^+(r) \cup B^-(r)$ we denote the head and (pos., resp. neg.) body of rule r. Rules with $|H(r)|$=1 and $B(r)$=$\emptyset$ are *facts* and rules with $H(r)$=$\emptyset$ *constraints*. A rule r is *positive*, if "$\texttt{not}$" does not occur in it, and *normal*, if $|H(r)| \leq 1$. A program Π is *positive* (resp. *normal*) if all its rules are positive (resp., normal). We omit "extended" in what follows and refer to EDLPs as DLPs etc.

Literals (resp. rules, programs) are *ground* if they are variable-free. Non-ground rules (resp. programs) amount to their *ground instantiation*, i.e., all rules obtained by substituting variables with constants from the (implicit) language.

A ground program Π is *head-cycle free* [1], if no literals $l \neq l'$ occurring in the same rule head mutually depend on each other by positive recursion; Π is stratified [20, 21], if no literal l depends by recursion through negation on itself.

Recall that the *answer set semantics* [7] for DLPs is as follows. Denote by $Lit(\Pi)$ the set of all ground literals for a program Π. Then, S is an *answer set* of Π, if S is a minimal (under $\subseteq$) consistent[4] set $S \subseteq Lit(\Pi)$ satisfying all rules in the reduct Π^S, which contains all rules $h_1 \;\texttt{v}\; \ldots \;\texttt{v}\; h_l \,\texttt{:-}\, b_1,\;\ldots,\;b_m$ for all ground instances of rules (1) in Π such that $S \cap B^-(r) = \emptyset$.

3 Meta-Interpreter Transformation

As mentioned above, a rewriting of a given program Π_{check} to a program Π'_{check} for integrating a guess and a check part into a single program is tricky in general. The

[3] See http://www.kr.tuwien.ac.at/staff/axel/guessncheck/ (forthcoming).
[4] For our concerns, we disregard a possible inconsistent answer set.

working of the answer set semantics is not easy to be emulated in Π'_{check}, since essentially we lack negation in Π'_{check}: Upon a "guess" S for an answer set of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$, the reduct Π^S_{solve} is not-free. Thus, contrary to Π_{check}, there is no possibility to consider varying guesses for the value of negated atoms in Π'_{check} in combination with one guess for the negated atoms in Π_{guess} in the combined program Π_{solve}. On the other hand, if there is no disjunction in Π'_{check} then Π_{solve} is Horn; thus, its answer sets can be guessed and checked in NP.

This leads us to consider an approach in which the program Π'_{check} is constructed by the use of meta-interpretation techniques [16, 2, 3]: the idea is that a program Π is represented by a set of facts, $F(\Pi)$, which is input to a fixed program Π_{meta}, the meta-interpreter, such that the answer sets of $\Pi_{meta} \cup F(\Pi)$ correspond to the answer sets of Π. Note that existing meta-interpreters are normal logic programs, and can not be used for our purposes for the reasons explained above; we have to construct a novel meta-interpreter which is essentially not-free and contains disjunction. To this end, we exploit the following characterization of (consistent) answer sets for HDLPs:

Theorem 1 (cf. [1]). *For any ground HDLP Π, a consistent $S \subseteq Lit(\Pi)$ is an answer set of Π iff (1) S satisfies Π and (2) there is a function $\phi : Lit(\Pi) \rightarrow \{0, 1, 2, \ldots\}$ such that for each literal $l \in S$, there is a rule $r \in \Pi$ with (a) $B^+(r) \subseteq S$, (b) $B^-(r) \cap S = \emptyset$, (c) $l \in H(r)$, (d) $S \cap (H(r) \setminus \{l\}) = \emptyset$, and (e) $\phi(l') < \phi(l)$ for each $l' \in B^+(r)$.*

Theorem 1 will now serve as a basis for a transformation from a given HDLP Π to a DLP $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$ such that $tr(\Pi)$ fulfills the properties **T1–T4**:

Input representation $F(\Pi)$ As input for the meta-interpreter Π_{meta} below, we choose the following representation $F(\Pi)$ of the propositional program Π.

We assume that each rule r has a unique name $n(r)$ as usual; for convenience, we identify r with $n(r)$. For any rule $r \in \Pi$, we set up in $F(\Pi)$ the following facts:

$\texttt{lit}(\texttt{h}, l, r)$. $\texttt{atom}(l, \lvert l \rvert)$.	for each literal $l \in H(r)$,
$\texttt{lit}(\texttt{p}, l, r)$.	for each literal $l \in B^+(r)$,
$\texttt{lit}(\texttt{n}, l, r)$.	for each literal $l \in B^-(r)$.

While the facts for predicate lit obviously encode the rules of Π, the facts for predicate atom indicate whether a literal is classically positive or negative. We only need this information for head literals; this will be further explained below.

Meta-Interpreter Π_{meta} We construct our meta-interpreter program Π_{meta}, which in essence is a positive disjunctive program, in a sequence of several steps. They center around checking whether a guess for an answer set $S \subseteq Lit(\Pi)$, encoded by a predicate $\texttt{inS}(\cdot)$, is an answer set of Π by testing the criteria of Theorem 1. The steps of the transformation cast the conditions of the theorem into rules of Π_{meta}, and provide auxiliary machinery for this aim.

Step 1 We add the following preprocessing rules:

```
1: rule(L,R) :- lit(h,L,R), not lit(p,L,R), not lit(n,L,R).
2: ruleBefore(L,R) :- rule(L,R), rule(L,R1), R1 < R.
3: ruleAfter(L,R) :- rule(L,R), rule(L,R1), R < R1.
4: ruleBetween(L,R1,R2) :- rule(L,R1), rule(L,R2), rule(L,R3),
                           R1 < R3, R3 < R2.
```

```
5:  firstRule(L, R) :- rule(L, R), not ruleBefore(L, R).
6:  lastRule(L, R) :- rule(L, R), not ruleAfter(L, R).
7:  nextRule(L, R1, R2) :- rule(L, R1), rule(L, R2), R1 < R2,
                          not ruleBetween(L, R1, R2).

8:  before(HPN, L, R) :- lit(HPN, L, R), lit(HPN, L1, R), L1 < L.
9:  after(HPN, L, R) :- lit(HPN, L, R), lit(HPN, L1, R), L < L1.
10: between(HPN, L, L2, R) :- lit(HPN, L, R), lit(HPN, L1, R),
                             lit(HPN, L2, R), L < L1, L1 < L2.
11: next(HPN, L, L1, R) :- lit(HPN, L, R), lit(HPN, L1, R), L < L1,
                          not between(HPN, L, L1, R).
12: first(HPN, L, R) :- lit(HPN, L, R), not before(HPN, L, R).
13: last(HPN, L, R) :- lit(HPN, L, R), not after(HPN, L, R).

14: hlit(L) :- rule(L, R).
```

Lines 1 to 7 fix an enumeration of the rules in Π from which a literal l may be derived, assuming a given order $<$ on rule names (e.g. in DLV [11], built-in lexicographic order; $<$ can also be easily generated using guessing rules). Note that under answer set semantics, we need only to consider rules where the literal l to prove does not occur in the body. Next, lines 8 to 13 fix enumerations of $H(r)$, $B^+(r)$ and $B^-(r)$ for each rule. The final line 14 collects all literals that can be derived from rule heads. Note that lines 1-14 plus $F(\Pi)$ form a stratified program, which has a single answer set (cf. [20, 21]).

Step 2 We add rules which "guess" a candidate answer set $S \subseteq Lit(\Pi)$ and a total ordering phi on S corresponding with the function ϕ in Theorem 1.(2).

```
15: inS(L) v ninS(L):-hlit(L).
16: ninS(L):-lit(pn, L, R), not hlit(L).                for each pn ∈ {p, n}
17: notok:-inS(L), inS(NL), L != NL, atom(L, A), atom(NL, A).
18: phi(L, L1) v phi(L1, L):-inS(L), inS(L1), L < L1.
19: phi(L, L2):-phi(L, L1), phi(L1, L2).
```

Line 15 focuses the guess of S to literals occurring in some relevant rule head in Π; other literals can not belong to S (line 16). Line 17 then checks whether S is consistent, deriving a new distinct atom notok otherwise. Line 18 guesses a strict total order phi on inS where line 19 guarantees transitivity; note that minimality of answer sets prevents that phi is cyclic, i.e., that phi(L, L) holds.

In the subsequent steps, we check whether S and phi violate the conditions of Theorem 1 by deriving the distinct atom notok in case, indicating that S is not an answer set or phi does not represent a proper function ϕ.

Step 3 Corresponding to condition 1 in Theorem 1, notok is derived whenever there is an unsatisfied rule by the following program part:

```
20: allInSUpto(p, Min, R):-inS(Min), first(p, Min, R).
21: allInSUpto(p, L1, R):- inS(L1), allInSUpto(p, L, R), next(p, L, L1, R).
22: allInS(p, R):-allInSUpto(p, Max, R), last(p, Max, R).

23: allNinSUpto(hn, Min, R):-ninS(Min), first(hn, Min, R).
24: allNinSUpto(hn, L1, R):- ninS(L1), allNinSUpto(hn, L, R),      for each
                             next(hn, L, L1, R).                    hn ∈ {h, n}
25: allNinS(hn, R):-allNinSUpto(hn, Max, R), last(hn, Max, R).
```

```
26: hasHead(R):-lit(h,L,R).
27: hasPBody(R):-lit(p,L,R).
28: hasNBody(R):-lit(n,L,R).
29: allNinS(h,R):-lit(HPN,L,R), not hasHead(R).
30: allInS(p,R):-lit(HPN,L,R), not hasPBody(R).
31: allNinS(n,R):-lit(HPN,L,R), not hasNBody(R).
32: notok:-allNinS(h,R), allInS(p,R), allNinS(n,R), lit(HPN,L,R).
```

These rules compute by iteration over $B^+(r)$ (resp. $H(r)$, $B^-(r)$) for each rule r, whether for all positive body (resp. head and weakly negated body) literals in rule r `inS` holds (resp. `ninS` holds) (lines 20 to 25). Here, empty heads (resp. bodies) are interpreted as unsatisfied (resp. satisfied), cf. lines 26 to 31. The final rule 32 fires exactly if one of the original rules from Π is unsatisfied.

Step 4 We derive `notok` whenever there is a literal $l \in S$ which is not provable by any rule r wrt. `phi`. This corresponds to checking condition 2 from Theorem 1.

```
33: failsToProve(L,R):-rule(L,R), lit(p,L1,R), ninS(L1).
34: failsToProve(L,R):-rule(L,R), lit(n,L1,R), inS(L1).
35: failsToProve(L,R):-rule(L,R), rule(L1,R), inS(L1), L1 != L, inS(L).
36: failsToProve(L,R):-rule(L,R), lit(p,L1,R), phi(L1,L).
37: allFailUpto(L,R):-failsToProve(L,R), firstRule(L,R).
38: allFailUpto(L,R1):- failsToProve(L,R1), allFailUpto(L,R),
                        nextRule(L,R,R1).
39: notok:-allFailUpto(L,R), lastRule(L,R), inS(L).
```

Lines 33 and 34 check whether condition 2.(a) or (b) are violated, i.e. some rule can only prove a literal if its body is satisfied. Condition 2.(d) is checked in line 35, i.e. r fails to prove l if there is some $l' \neq l$ such that $l' \in H(r) \cap S$. Violations of condition 2.(e) are checked in line 36. Finally, lines 37 to 39 derive `notok` if all rules fail to prove some literal $l \in S$ by iterating over all rules with $l \in H(r)$ using the order from Step 1. Thus, condition 2.(c) is implicitly checked.

Step 5 Whenever `notok` is derived, indicating a wrong guess, then we apply a saturation technique as in [6, 12] to some other predicates, such that a canonical set Ω results. This set turns out to be an answer set iff no guess for S and ϕ works out, i.e., Π has no answer set. In particular, we saturate the predicates `inS`, `ninS`, and `phi` by the following rules:

```
40: phi(L,L1):-notok, hlit(L), hlit(L1).
41: inS(L):-notok, hlit(L).
42: ninS(L):-notok, hlit(L).
```

Intuitively, by these rules, any answer set containing `notok` is "blown up" to an answer set Ω containing all possible guesses for `inS`, `ninS`, and `phi`.

3.1 Answer Set Correspondence

Let $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$, where $F(\Pi)$ and Π_{meta} are the input representation and meta-interpreter as defined above. Clearly, $tr(\Pi)$ satisfies property **T3**, and as easily checked, $tr(\Pi)$ is modular. Moreover, $\neg$ does not occur in $tr(\Pi)$ and `not` only stratified. The latter is not applied to literals depending on disjunction; it thus occurs only in the deterministic part of $tr(\Pi)$, i.e. **T4** holds.

To establish **T1** and **T2**, we define the literal set Ω as follows:

Definition 1. *Let Π^i_{meta} be the set of rules in Π_{meta} established in Step $i \in \{1, \ldots, 5\}$. For any program Π, let $\Pi_\Omega = F(\Pi) \cup \bigcup_{i\in\{1,3,4,5\}} \Pi^i_{meta} \cup \{\texttt{notok.}\}$. Then, Ω is defined as the answer set of Π_Ω.*

The fact that Π_Ω is a stratified normal logic program without $\neg$ and constraints, which as well-known has a single answer set, yields the following lemma.

Lemma 1. *Ω is well-defined and uniquely determined by Π.*

Theorem 2. *For any given HDLP Π the following holds for $tr(\Pi)$:*

1. *$tr(\Pi)$ has some answer set, and $S' \subseteq \Omega$ for any answer set S' of $tr(\Pi)$.*
2. *S is an answer set of Π if and only if there exists an answer set S' of $tr(\Pi)$ such that $S = \{l \mid \texttt{inS}(l) \in S'\}$ and $\texttt{notok} \notin S'$.*
3. *Π has no answer set if and only if $tr(\Pi)$ has the unique answer set Ω.*

The following proposition is not difficult to establish.

Proposition 1. *Given Π, the transformation $tr(\Pi)$, as well as the ground instantiation of $tr(\Pi)$, is computable in* LOGSPACE *(thus in polynomial time).*

Note that $tr(\Pi)$ is not polynomial faithful modular (PFM) in the sense of [9]: (i) **T1** does not claim a strict one-to-one correspondence between the answer sets of Π and $tr(\Pi)$. Indeed, $tr(\Pi)$ might have several answer sets corresponding to a particular answer set S of Π, reflecting different possible guesses for ϕ. (ii) Faithfulness as in [9] conflicts with property **T2**.

As noticed above, $tr(\Pi)$ uses weak negation only stratified and in a deterministic part of the program; we can easily eliminate it by computing in the transformation the complement of each predicate accessed through `not` and providing it in $F(\Pi)$ as facts; we then obtain a positive program. (The built-in predicates $<$ and != can be eliminated similarly if desired.) However, this modified transformation is not modular. As shown next, this is not incidental.

Proposition 2. *There is no modular transformation $tr'(\Pi)$ from HDLPs to DLPs satisfying* **T1**, **T2** *and* **T3** *such that $tr'(\Pi)$ is a positive program.*

Proof. Assuming that such a transformation $tr'(\Pi)$ exists, we derive a contradiction. Let $\Pi_1 = \{\texttt{a} :- \texttt{not b.}\}$ and $\Pi_2 = \Pi_1 \cup \{\texttt{b.}\}$. Then, $tr'(\Pi_2)$ has some answer set S_2. Since $tr'(\cdot)$ is modular, $tr'(\Pi_1) \subseteq tr'(\Pi_2)$ holds and thus S_2 satisfies each rule in $tr'(\Pi_1)$. Hence, S_2 contains some answer set S_1. By **T1**, $\texttt{inS(a)} \in S_1$ must hold, and hence $\texttt{inS(a)} \in S_2$. By **T1** again, it follows that Π_2 has an answer set S such that $\texttt{a} \in S$. But the single answer set of Π_2 is $\{\texttt{b}\}$, a contradiction. □

We remark that Proposition 2 remains true if **T1** is generalized such that the answer set S of Π corresponding to S' is given by $S = \{l \mid S' \models \Phi(l)\}$, where $\Phi(x)$ is a monotone query (e.g., computed by a normal positive program without constraints). Moreover, if a successor predicate $\texttt{next(X, Y)}$ and predicates $\texttt{first(X)}$ and $\texttt{last(X)}$ for the constants are available (on a finite universe, resp. the constants in Π and rule names), then the negation of the non-input predicates accessed through `not` can be computed by a positive normal program, since such programs capture polynomial time computability by well-known results on the expressive power of Datalog [18]; thus, negation of input predicates in $F(\Pi)$ is sufficient in this case.

3.2 Optimizations

Π_{meta} can be modified in several respects. We discuss here some modifications which, though not necessarily shrinking size of the transformation, intuitively prune the search of an answer set solver applied to $tr(\Pi)$. The extended paper considers further ones.

(OPT1) *Give up modularity* If we sacrifice modularity (i.e. that $tr(\Pi) = \bigcup_{r\in\Pi} tr(r)$), and allow that Π_{meta} partly depends on the input, then we can circumvent the iterations in Step 3 and part of Step 1 as follows: We substitute Step 3 by rules

$$\texttt{notok} :- \texttt{ninS}(h_1), \ldots, \texttt{ninS}(h_l),\ \texttt{inS}(b_1), \ldots, \texttt{inS}(b_m), \texttt{ninS}(b_{m+1}), \ldots \texttt{ninS}(b_n). \quad (2)$$

for each rule r in Π of form (1). These rules can be efficiently generated in parallel to $F(\Pi)$. Lines 8 to 13 of Step 1 then can also be dropped.

We can even refine this further. For any normal rule $r \in \Pi$ with $|H(r)| = 1$ which has a satisfied body, we can force the guess of h: we replace (2) by

$$\texttt{inS}(h) :- \texttt{inS}(b_1),\ \ldots,\ \texttt{inS}(b_m), \texttt{ninS}(b_{m+1}),\ \ldots\ \texttt{ninS}(b_n). \quad (3)$$

In this context, since constraints only serve to "discard" unwanted models but cannot prove any literal, we can ignore them during input generation $F(\Pi)$; rule (2) is sufficient. Note that dropping input representation $\texttt{lit}(\texttt{n}, l, c)$. for literals only occurring in the negative body of constraints but nowhere else in Π requires some care. Such l can be removed by simple preprocessing, though.

(OPT2) *Optimize guess of order* We only need to guess and check the order ϕ for literals L, L' if they allow for cyclic dependency, i.e., they appear in the heads of rules within the same strongly connected component of the program wrt. S.[5] These dependencies wrt. S are easily computed:

```
dep(L, L1):-lit(h, L, R), lit(p, L1, R), inS(L), inS(L1).
dep(L, L2):-lit(h, L, R), lit(p, L1, R), dep(L1, L2), inS(L).
cyclic:-dep(L, L1), dep(L1, L).
```

The guessing rules for ϕ (line 18 and 19) are then be replaced by:

```
phi(L, L1) v phi(L, L1):-dep(L, L1), dep(L1, L), L < L1, cyclic.
phi(L, L2):-phi(L, L1), phi(L1, L2), cyclic.
```

Moreover, we add the new atom `cyclic` also to the body of the rules where `phi` appears (lines 36,40) to check `phi` only if Π has *any* cyclic dependencies wrt. S.

4 Integrating Guess and co-NP Check Programs

A general method for solving NP problems using answer set programming is given by the so called "guess and check" paradigm: First a (possibly disjunctive) program is used to guess a set of candidate solutions, and then rules and constraints are added which eliminate unwanted solutions. DLPs allow for the formulation of such problems in a very intuitive way (e.g. solutions of 3-colorability, deterministic planning, etc.) if checking is easy (often polynomial), such as checking whether no adjacent nodes have the same color, a course of deterministic actions reaches a certain goal, etc. For instance, given a graph as a set of facts of the form $\texttt{node}(x)$. and $\texttt{edge}(x, y)$. we can write a simple DLP which guesses and checks all possible 3-colorings as follows:

[5] Similarly, in [1] $\phi : Lit(\Pi) \to \{1, \ldots, r\}$ is only defined for a range r bound by the longest acyclic path in any strongly connected component of the program.

$$
\begin{array}{ll}
\texttt{col(red, X) v col(green, X) v col(blue, X) :- node(X).} & \} \ \textit{Guess} \\
\texttt{:-edge(X, Y), col(C, X), col(C, Y).} & \} \ \textit{Check}
\end{array}
$$

However, encoding problems where the check is in co-NP but not known to be polynomial (or in NP) is not always obvious (e.g., for conformant planning [4], or minimal update answer sets [5]). A simple, common workaround is to write two programs:

(i) a normal LP or HDLP Π_{guess}, which guesses some solution;
(ii) a HDLP (equivalently, normal LP) Π_{check} which encodes the co-NP check,

and proceed as follows: First compute, one by one, the candidate solutions $S_1, S_2, \ldots$ as answer sets of Π_{guess}; then, pipe each S_i as input to Π_{check}; finally, output S_i if $\Pi_{check} \cup S_i$ has no answer set.

By the computational power of full disjunctive logic programs (Σ_2^P [6]), we know that such problems can also be expressed by a single EDLP, Π_{solve}. In the following, we show how our transformation tr resp. tr_{Opt} from above can be used to automatically combine Π_{guess} and Π_{check} into a single program.

We assume that the set $Lit(\Pi_{guess})$ is a Splitting Set [13] of $\Pi_{guess} \cup \Pi_{check}$, i.e. no head literal from Π_{check} occurs in Π_{guess}. This can be easily achieved by introducing new predicate names, e.g., $\texttt{p}'$ for a predicate $\texttt{p}$, and adding a rule $\texttt{p}'(t)\texttt{:-}\texttt{p}(t)$ in case. Each rule r in Π_{check} is of the form

$$
\begin{array}{lll}
h_1 \ \texttt{v} \ \cdots \ \texttt{v} \ h_l & \texttt{:-} & bc_1, \ \ldots, \ bc_m, \ \texttt{not} \ bc_{m+1}, \ \ldots, \ \texttt{not} \ bc_n \\
 & & bg_1, \ \ldots, \ bg_p, \ \texttt{not} \ bg_{p+1}, \ \ldots, \ \texttt{not} \ bg_q.
\end{array} \tag{4}
$$

where the bg_i are the body literals defined in Π_{guess}. We write $\mathsf{B}_{guess}(r)$ for $bg_1, \ldots,$ bg_p, $\texttt{not}\ bg_{p+1}, \ldots,$$\texttt{not}\ bg_q$. We now define a new check program.

Program $\mathbf{\Pi}'_{check}$ contains the following rules and constraints:

1. The facts $F(\Pi_{check})$ in a conditional version: For each $r \in \Pi_{check}$ of form (4),

$$
\begin{array}{rlll}
\texttt{lit}(\texttt{h}, l, r) & \texttt{:-}\ \mathsf{B}_{\texttt{guess}}(r). & \texttt{atom}(l, |l|). & \text{for each } l \in H(r); \\
\texttt{lit}(\texttt{p}, bc_i, r) & \texttt{:-}\ \mathsf{B}_{\texttt{guess}}(r). & & \text{for each } i \in \{1, \ldots, m\}; \\
\texttt{lit}(\texttt{n}, bc_j, r) & \texttt{:-}\ \mathsf{B}_{\texttt{guess}}(r). & & \text{for each } j \in \{m+1, \ldots, n\}.
\end{array}
$$

2. each rule in Π_{meta} (where for the optimized version, in (2) and (3) $\mathsf{B}_{guess}(r)$ is added to the bodies);
3. finally, a constraint $\texttt{:-}\ \texttt{not notok}$. This will eliminate all answer sets S such that $\Pi_{check} \cup S$ has an answer set.

The union of Π_{guess} and Π'_{check} then amounts to the desired integrated encoding Π_{solve}, which is expressed by the following result.

Theorem 3. *For Π_{guess} and Π_{check}, the answer sets S' of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$ correspond 1-1 with the answer sets S of Π_{guess} s.t. $\Pi_{check} \cup S$ has no answer set.*

Note that integrating guess and check programs Π_{guess} and Π_{check}, respectively, to succeed iff $\Pi_{check} \cup S$ has *some* answer set, is easy. After ensuring the Splitting Set property (if needed), simply take $\Pi_{solve} = \Pi_{guess} \cup \Pi_{check}$; its answer sets correspond on the predicates in Π_{guess} to the desired solutions.

5 Applications

We now exemplify the use of our transformation for two Σ_2^P-complete problems, which thus involve co-NP-complete solution checking: one is about Quantified Boolean formulas (QBFs) with one quantifier alternation, which are well-studied in Answer Set Programming, and the other about conformant planning [4, 22]. Further examples of such problems can be found e.g. in [6, 5, 11] (and solved similarly). However, note that our method is applicable to *any* checks encoded by inconsistency of some HDLP program; co-NP-hardness is not a prerequisite.

5.1 Quantified Boolean Formulas

Given a QBF $F = \exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n \Phi$, where $\Phi = c_1 \vee \cdots \vee c_k$ is a propositional formula over $x_1, \ldots, x_m, y_1, \ldots, y_n$ in disjunctive normal form, i.e. each $c_i = \ell_{i,1} \wedge \cdots \wedge \ell_{i,i_l}$ and $|\ell_{i,j}| \in \{x_1, \ldots, x_m, y_1 \ldots, y_n\}$, compute the assignments to the variable $x_1, \ldots, x_m$ which witness that F evaluates to true.

Intuitively, this problem can be solved by "guessing and checking" as follows:

(QBF_g) Guess a truth assignment for the variables $x_1, \ldots, x_m$.
(QBF_c) Check whether this assignment satisfies Φ for all assignments of $y_1, \ldots, y_n$.

Both parts can be encoded by very simple HDLPs:

QBF_g :

x_1 v $-x_1$. ... x_m v $-x_m$.

QBF_c :

y_1 v $-y_1$. ... y_n v $-y_n$.

:- $\ell_{1,1}, \ldots, \ell_{1,1_l}$. ... :- $\ell_{k,1}, \ldots, \ell_{k,k_l}$.

Obviously, for any answer set S of QBF_g, representing an assignment to $x_1, \ldots, x_n$, the program $QBF_c \cup S$ has no answer set thanks to the constraints, iff every assignment for $y_1, \ldots, y_n$ satisfies formula Φ then. By the method sketched, we can now automatically generate a single program QBF_{solve} integrating the guess and check programs (cf. Footnote 3). Note that the customary (but tricky) saturation technique to solve this problem (cf. [6, 11]) is fully transparent to the non-expert.

5.2 Conformant planning

Loosely speaking, planning is the problem to find a sequence of actions $P = \alpha_1, \alpha_2, \ldots, \alpha_n$, a *plan*, which takes a system from an initial state s_0 to a state s_n in which a goal (often, given by an atom g) holds, where a state s is described by values of fluents, i.e., predicates which might change over time. *Conformant planning* [8] is concerned with finding a plan P which works under all contingencies that may arise from incomplete information about the initial state and/or nondeterministic action effects, which is in Σ_2^P under certain restrictions, cf. [4, 22]. Hence, the problem can be solved with a guess and (co-NP) check strategy.

As an example, we consider a simplified version of the well-known "*Bomb in the Toilet*" planning problem (cf. [4, 17]): We have been alarmed that a possibly armed bomb is in a lavatory which has a toilet bowl. Possible actions are dunking the bomb into the bowl and flushing the toilet. After just dunking, the bomb may be disarmed or not; only flushing the toilet guarantees that it is really disarmed.

Using the following guess and check programs $Bomb_g$ and $Bomb_c$, respectively, we can compute a plan for having the bomb disarmed by two actions:

$Bomb_g$:

```
% Timestamps:
time(0). time(1).
%Guess a plan:
dunk(T) v -dunk(T):-time(T).
flush(T) v -flush(T):-time(T).
% Forbid concurrent actions:
:- flush(T), dunk(T).
```

$Bomb_c$:

```
% Initial state:
armed(0) v -armed(0).
% Frame Axioms:
armed(T1):-armed(T), time(T),
            not -armed(T1), T1 = T + 1.
dunked(T1):-dunked(T), T1 = T + 1.
% Effect of dunking:
dunked(T1):-dunk(T), T1 = T + 1.
armed(T1) v -armed(T1):-dunk(T),
                         armed(T), T1 = T + 1.
% Effect of flushing:
 -armed(T1):-flush(T), dunked(T), T1 = T + 1.
% Check goal in stage 2:
 :- not armed(2).
```

$Bomb_g$ guesses all candidate plans $P = \alpha_1, \alpha_2$, using time points for action execution, while $Bomb_c$ checks whether any such plan P is conformant for the goal $g =$ $\texttt{not armed}(2)$. Here, absence of $\texttt{armed}(t)$ is viewed as $\texttt{-armed}(t)$, i.e. CWA is used, which saves a negative frame axiom on `-armed`. The final constraint eliminates a plan execution iff it reaches the goal; thus, $Bomb_c$ has no answer set iff the plan P is conformant. Answer set $S = \{\texttt{time}(0), \texttt{time}(1), \texttt{dunk}(0), \texttt{flush}(1)\}$ of $Bomb_g$ corresponds to the (single) conformant plan P= `dunk`, `flush` for goal $\texttt{not armed}(2)$.

By our general method, $Bomb_g$ and $Bomb_c$ can be integrated automatically into a single program $Bomb_{plan} = Bomb_g \cup {Bomb_c}'$ (cf. Footnote 3). It has a single answer set, corresponding to the single conformant plan P= `dunk`, `flush` as desired.

Note that our rewriting method is more generally applicable than the encoding for conformant planning proposed by Leone *et al.* [12] who require that state transitions are specified by a positive constraint-free LP. Our method can still safely be used in presence of negation and constraints, provided action execution always leads to a consistent successor state (cf. [4, 22] for a discussion).

6 Experiments

We have conducted some experiments to get an idea about the performance of our automatically integrated encodings (even though this was not the major concern). To this end, we made comparisons to hand-written integrated encodings (in particular, for QBF evaluation using an ad hoc encoding from [11]) and to interleaved guess and check programs (in particular, for conformant planning on variants of the Bomb in the Toilet problem, executed on the DLV$^{\mathcal{K}}$ planning system [4]). The results are shown in Table 1.

We have considered some random QBF instances with n existential and n universal variables, denoted as QBF-n. For conformant planning, we have compared the integrated encodings for some "Bomb in the Toilet" problems described in [4], where the names BTC(i), BTUC(i) are defined, against the DLV$^{\mathcal{K}}$ planning system [4], which implements conformant planning by interleaved calls to separate guess and check programs. The columns in Table 1 report times for encodings using the basic approach ($meta$), optimizations $OPT1$, $OPT2$ and both (OPT), respectively, compared with the ad hoc encoding for QBFs and interleaved conformant plan computation using DLV$^{\mathcal{K}}$; a

	$adhoc$ [11]	$meta$	$OPT1$	$OPT2$	OPT
QBF-4	0.01s	0.15s	0.10s	0.08s	0.06s
QBF-6	0.01s	1.08s	0.36s	0.17s	0.08s
QBF-8	0.01s	10.42s	1.43s	0.46s	0.10s
QBF-10	0.01s	60.74s	2.65s	1.32s	0.10s
QBF-12	0.01s	-	-	5.59s	0.11s
QBF-16	0.08s	-	-	-	0.50s

Average times for 10 randomly chosen instances per size.

	$\mathtt{DLV}^{\mathcal{K}}$ [4]	$meta$	$OPT1$	$OPT2$	OPT
BTC(2)	0.01s	1.16s	0.80s	0.15s	0.08s
BTC(3)	0.11s	9.33s	9.25s	8.18s	4.95s
BTC(4)	4.68s	71.3s	67.8s	333s	256s
BTUC(2)	0.01s	6.38s	6.26s	0.22s	0.17s
BTUC(3)	1.78s	-	-	28.12s	13.0s
BTUC(4)	577s	-	-	-	2322s

Table 1. Experimental results for QBF (left) and Conformant Planning (right) for Bomb in Toilet

dash marks exceeding a time limit of 300s (QBFs), resp. 4000s (conformant planning). We used DLV [6] as a platform since other disjunctive ASP engines, in particular GNT,[7] were significantly slower on all tested instances. More details and experiments are given in the extended paper (cf. Footnote 3).

Clearly, the performance of the automatic integrated encodings was expected to stay behind the other methods. Interestingly, for the QBF problem, the performance of our optimized translation stays within reach of the ad hoc encoding in [11] for small instances. For the planning problems, the integrated encodings tested still stay behind the interleaved computation of $\mathtt{DLV}^{\mathcal{K}}$.

The results obtained using DLV show that the "guess and saturate" strategy in our approach benefits a lot from optimizations, but it might depend on the structure of Π_{guess} and Π_{check}, as well as on the heuristics used by DLV, which modifications yield gains. We strongly believe that there is room for further improvements both on the translation and for the underlying DLV engine. We emphasize that the strength of our approach appears if an integrated ad hoc encoding is non-obvious. Then, by our method such an encoding can be generated automatically from separate guess and check programs, which are often easy to formalize, while a manual integrated encoding may be difficult to find (as in the case of conformant planning or minimal update answer sets [5]).

7 Conclusion

We presented a method for rewriting a head-cycle free (extended) disjunctive logic program (HDLP) Π into a stratified disjunctive logic program without constraints $tr(\Pi)$, such that their answer sets correspond and a designated answer set of $tr(\Pi)$ indicates inconsistency of Π. Moreover, we showed how to use this method for automatically integrating a guess and separate check program for a co-NP property (expressed by inconsistency of an HDLP), into an equivalent single (extended) disjunctive logic program. This reconciles pragmatic problem solving with the natural "guess and check" resp. "Generate/Define/Test" approach in Answer Set Programming [11, 14], in case a single program expressing the problem is difficult to write. In particular, it relieves the ASP user from using tricky saturation techniques, as customary e.g. for QBF evaluation.

We consider our work as an initial step for further research on automatic integration of guess and check encodings which exploits the full expressive power of DLPs: Integrated encodings like those considered are infeasible in less expressive frameworks such as propositional SAT solving or normal logic programming. However, while ad hoc encodings for specific problems are available, general methods in this direction were still missing.

[6] http://www.dlvsystem.com

[7] http://www.tcs.hut.fi/Software/gnt/

Several issues remain for further work. Our rewriting method currently applies to propositional programs. Thus, before transformation, the program should be instantiated. A more efficient extension of the method to non-ground programs is needed, as well as further improvements to the current transformations. Experimental results suggest that structural analysis of the guess and check programs might be valuable for this.

A further issue are alternative transformations, possibly tailored for certain classes of programs. Ben-Eliyahu and Dechter's work [1], on which we build, aimed at transforming HDLPs to SAT problems. It might be interesting to investigate whether related methods such as the one developed for ASSAT [15], which was recently generalized by Lee and Lifschitz [10] to disjunctive programs, can be adapted for our approach.

References

1. R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
2. J. Delgrande, T. Schaub, H. Tompits. plp: A generic compiler for ordered logic programs. *Proc. LPNMR'01*, LNCS 2173, pp. 411–415. Springer, 2001.
3. T. Eiter, W. Faber, N. Leone, G. Pfeifer. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory & Practice of Logic Progr.*, 3(4-5):463–498.
4. T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres. A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system. *Artif. Intell.*, 144(1-2):157-211, 2003.
5. T. Eiter, M. Fink, G. Sabbatini, H. Tompits. On properties of update sequences based on causal rejection. *Theory & Practice of Logic Prog.*, 2(6):721–777, 2002.
6. T. Eiter, G. Gottlob, H. Mannila. Disjunctive datalog. *ACM TODS*, 22(3):364–418, 1997.
7. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
8. R. Goldman and M. Boddy. Expressive planning and explicit knowledge. *Proc. 3rd Int'l Conf. on AI Planning and Scheduling (AIPS-96)*, pp. 110–117, 1996.
9. T. Janhunen. On the effect of default negation on the expressiveness of disjunctive rules. *LPNMR'01*, LNCS 2173, pp. 93–106. Springer, 2001.
10. J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. 19th Int'l Conf. on Logic Programming (ICLP-03)*, December 2003. To appear.
11. N. Leone, G. Pfeifer, W. Faber *et al.* The DLV system for knowledge representation and reasoning. Tech. Rep. INFSYS RR-1843-02-14, Information Sys. Institute, TU Wien, 2002.
12. N. Leone, R. Rosati, F. Scarcello. Enhancing answer set planning. *Proc. IJCAI-01 Workshop on Planning under Uncertainty & Incomplete Information*, pp. 33–42, 2001.
13. V. Lifschitz and H. Turner. Splitting a logic program. *Proc. ICLP'94*, pp. 23–37, 1994.
14. V. Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138:39–54, 2002.
15. F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Proc. 18th National Conf. on Artificial Intelligence (AAAI-2002)*, 2002.
16. V.W. Marek and J.B. Remmel. On the Expressibility of stable logic programming. *Proc. LPNMR'01*, LNCS 2173, pp. 107–120. Springer, 2001.
17. D. McDermott. A critique of pure reason. *Computational Intelligence*, 3:151–237, 1987.
18. C. H. Papadimitriou. A note on the expressive power of Prolog. *Bulletin of the EATCS*, 26:21–23, 1985.
19. A. Provetti and T.C. Son, editors. *Proc. AAAI 2001 Spring Symposium on Answer Set Programming*, Stanford, CA, March 2001. Workshop Technical Report SS-01-01, AAAI Press.
20. T. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.
21. T. Przymusinski. Stable semantics for disjunctive programs. *New Gen. Comp.*, 9, 1991.
22. H. Turner. Polynomial-length planning spans the Polynomial Hierarchy. *Proc. 8th European Conf. on Artificial Intelligence (JELIA 2002)*, LNCS 2424, pp. 111–124. Springer, 2002.