I N F S Y S
R E S E A R C H
R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME

ARBEITSBEREICH WISSENSBASIERTE SYSTEME

TOWARDS AUTOMATED INTEGRATION OF GUESS AND CHECK PROGRAMS IN ANSWER SET PROGRAMMING: A META-INTERPRETER AND APPLICATIONS

Thomas Eiter   Axel Polleres

INFSYS RESEARCH REPORT 1843-04-01
JANUARY 2004; JANUARY 2005

Institut für Informationssysteme
AB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstrassße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TU
WIEN

INFSYS RESEARCH REPORT
INFSYS RESEARCH REPORT 1843-04-01, JANUARY 2004; JANUARY 2005

TOWARDS AUTOMATED INTEGRATION OF GUESS AND CHECK PROGRAMS IN ANSWER SET PROGRAMMING: A META-INTERPRETER AND APPLICATIONS

Thomas Eiter[1] Axel Polleres[2]

Abstract. Answer set programming (ASP) with disjunction offers a powerful tool for declaratively representing and solving hard problems. Many NP-complete problems can be encoded in the answer set semantics of logic programs in a very concise and intuitive 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, and captures a class of problems at the second level of the polynomial hierarchy. While these problems also have a clear "guess and check" structure, finding an encoding in a DLP reflecting this structure may sometimes be a non-obvious task, in particular if the "check" itself is a co-NP-complete problem; usually, such problems are solved by interleaving separate guess and check programs, where the check is expressed by inconsistency of the check program. In this paper, we present general transformations of head-cycle free (extended) disjunctive logic programs into stratified and positive (extended) disjunctive logic programs based on meta-interpretation techniques. The answer sets of the original and the transformed program are in simple correspondence, and, moreover, inconsistency of the original program is indicated by a designated answer set of the transformed program. Our transformations facilitate the integration of separate "guess" and "check" programs, which are often easy to obtain, 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.
Keywords: answer set programming, disjunctive logic programs, guess and check paradigm, meta-interpretation, automated program synthesis

---

[1]Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, 1040 Wien, Austria. E-mail: eiter@kr.tuwien.ac.at.
[2]Institut für Informatik, Universität Innsbruck, Technikerstraße 21, A-6020 Innsbruck, Austria. E-mail: axel.polleres@deri.org

Acknowledgements: This work was partially supported by FWF (Austrian Science Funds) under projects P14781 and Z29-N04, and Information Society Technologies program of the European Commission, Future and Emerging Technologies under the projects IST-2001-37004 (WASP) and IST-2001-33570 (INFOMIX).

To appear in Theory and Practice of Logic Programming (TPLP). Part of the results in this paper are contained, in preliminary form, in the Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004), Springer LNCS 2923, Ilkka Niemelä and Vladimir Lifschitz (eds), pp. 100-113 and in the informal Proc. of the Joint Conference on Declarative Programming (APPIA-GULP-PRODE) 2003.

Contents

1 Introduction

Answer set programming (ASP) [35, 15, 26, 29, 31], also called A-Prolog [1, 2, 16], is widely proposed as a useful tool for solving problems in a declarative manner, by encoding the solutions to a problem in the answer sets of a normal logic program. By well-known complexity results, in this way all problems with complexity in NP can be expressed and solved [39, 28]; see also [6].

A frequently considered example of an NP-complete problem which can be elegantly solved in ASP is Graph-3-Colorability, i.e., deciding whether some given graph G is 3-colorable. It is an easy exercise in ASP to write a program which determines whether a graph is 3-colorable. A straightforward encoding, following the "Guess and Check" [8, 23] respectively "Generate/Define/Test" approach [26], consists of two parts:

- A "guessing" part, which assigns nondeterministically each node of the graph one of three colors:

  ```
  col(red,X) v col(green,X) v col(blue,X) :- node(X).
  ```

- and a "checking" part, which tests whether no adjacent nodes have the same color:

  ```
  :-edge(X,Y), col(C,X), col(C,Y).
  ```

Here, the graph G is represented by a set of facts node (x) and edge (x, y). Each legal 3-coloring of G is a polynomial-size "proof" of its 3-colorability, and such a given proof can be validated in polynomial time. Furthermore, the answer sets of this program yield all legal 3-colorings of the graph G.

However, we 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 problems in co-NP are in NP, and hence that it is not always the case that a polynomial-size "proof" of a co-NP property P exists which can be verified in polynomial time. For such problems, we thus can not write a (polynomial-size propositional) normal logic program in ASP which guesses and verifies in its answer sets possible "proofs" of P. One such property, for instance, is the co-NP-complete property that a given graph is *not* 3-colorable. However, this and similar properties P can be dually expressed in ASP in terms of whether a normal logic program (equivalently, a head-cycle free disjunctive logic program [3]) Π_P has no answer set if and only if the property p holds.

Properties that are co-NP-complete often occur within the context of problems that reside in the class Σ_2^P, which is above NP in the polynomial time hierarchy [33]. In particular, the solutions of a Σ_2^P-complete problem can be typically singled out from given candidate solutions by testing a co-NP-complete property. Some well-known examples of such Σ_2^P-complete problems are the following ones, which will be further detailed in Section 6:

Quantified Boolean Formulas: Evaluating a Quantified Boolean formula (QBF) of the form $\exists X \forall Y \Phi(X, Y)$, where $\Phi(X, Y)$ is a disjunctive normal form over propositional variables $X \cup Y$. Here, a solution is a truth value assignment σ to the variables X such that the formula $\forall Y \Phi(\sigma(X), Y)$ evaluates to true, i.e., $\Phi(\sigma(X), Y)$ is a tautology. Given a candidate solution σ, the co-NP-complete property to check here is whether $\Phi(\sigma(X), Y)$ is a tautology.

Strategic Companies: Computing strategic companies sets [8, 23]. Roughly, here the problem is to compute, given a set of companies C in a holding, a minimal subset $S \subseteq C$ which satisfies some constraints concerning the production of goods and control of companies. Any such set is called strategic; Given a candidate solution S which satisfies the constraints, the co-NP-complete property to check here is the minimality, i.e., that no set $S' \subset S$ exists which also satisfies the constraints.

Conformant Planning: Computing conformant plans under incomplete information and nondeterministic action effects. Here the problem is to generate from a description of the initial state I, the planning goal G, and the actions α and their effects a sequence of actions (a plan) $P = \alpha_1, \ldots, \alpha_n$ which carries the agent from the initial state to a goal-fulfilling state under all contingencies, i.e., regardless of the precise initial state and how non-deterministic actions work out. Given a candidate solution in terms of an *optimistic* plan P, which works under *some* execution [10], the property to check is whether it works under *all* executions, i.e., whether it is conformant [17]. The latter problem is in co-NP, provided that executability of actions is polynomially decidable, cf. [10, 40].

This list can be extended, and further examples can be found, e.g., in [13, 12, 18, 38].

The problems described above can be solved using ASP in a two-step approach as follows:

1. Generate a candidate solution S by means of a logic program Π_{guess}.

2. Check the solution S by "running" another logic program Π_{check} (=Π_p) on S, such that $\Pi_{check} \cup S$ has no answer set if and only if S is a valid solution.

The respective programs Π_{check} can be easily formulated (cf. Section 6).

On the other hand, ASP with disjunction, i.e. full extended disjunctive logic programming, allows one to formulate problems in Σ_2^P in a *single* (disjunctive) program, since this formalism captures the complexity class Σ_2^P, cf. [6, 13]. Hence, efficient ASP engines such as DLV [23] or GNT [21] can be used to solve such programs directly in a one-step approach.

A difficulty here is that sometimes, an encoding of a problem in a single logic program (e.g., for the conformant planning problem above) may not be easy to find. This raises the issue whether there exists an (effective) possibility to *combine* separate Π_{guess} and Π_{check} programs into a single program Π_{solve}, such that this unified program computes the same set of solutions as the two-step process outlined above. A potential benefit of such a combination is that the space of candidate solutions might be reduced in the evaluation due to its interaction with the checking part. Furthermore, automated program optimization techniques may be applied which consider both the guess and check part as well as the interactions between them. This is not possible for separate programs.

The naive attempt of taking the union $\Pi_{guess} \cup \Pi_{check}$ unsurprisingly fails: indeed, each desired answer set of Π_{guess} would be eliminated by Π_{check} (assuming that, in a hierarchical fashion, Π_{check} has no rules defining atoms from Π_{guess}). Therefore, some program transformation is necessary. A natural question here is whether it is possible to rewrite Π_{check} to some other program Π'_{check} such that an integrated logic program $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$ is feasible, and, moreover, whether this can be done automatically.

From theoretical complexity results about disjunctive logic programs cf. [6, 13], one can infer that the program Π'_{check} should be truly disjunctive in general, i.e., not rewritable to an equivalent non-disjunctive program in polynomial time. This and further considerations (see Section 3) provide some evidence that a suitable rewriting of Π_{check} to Π'_{check} is not immediate.

In this paper, we therefore address this issue and present a generic method for constructing the program Π'_{check} by using a meta-interpreter approach. In particular, we make the following contributions:

(1) We provide a transformation $tr(\Pi)$ from propositional head-cycle-free [3] (extended) disjunctive logic programs (HDLPs) Π to disjunctive logic programs (DLPs), which enjoys the properties that the answer sets of $tr(\Pi)$ encode the answer sets of Π, if Π has some answer set, and that $tr(\Pi)$ has a canonical answer set otherwise which is easy to recognize. The transformation $tr(\Pi)$ is polynomial and modular in the sense of [19], and employs meta-interpretation of Π.

Furthermore, we describe variants and modifications of $tr(\Pi)$ aiming at optimization of the transformation. In particular, we present 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, the Strategic Companies problem, and conformant planning [17] under fixed polynomial plan length (cf. [10, 40]). Our method proves useful to loosen some restrictions of previous encodings, and to obtain disjunctive encodings for more general problem classes.

(4) We compare our approach on integrating separate guess and check programs experimentally against existing ad hoc encodings for QBFs and Strategic Companies and also applying it to conformant planning, where no such ad hoc encodings were known previously. For these experiments, we use `DLV` [23], a state-of-the-art Answer Set engine for solving DLPs. The results which we obtained reveal interesting aspects: While as intuitively expected, efficient ad hoc encodings have better performance than the synthesized integrated encodings in general, there are also cases where the performances scale similarly (i.e., the synthesized encoding is within a constant factor), or where even ad hoc encodings from the literature are outperformed.

Our results contribute to further the "Guess and Check" resp. "Generate/Define/ Test" paradigms for ASP, and fill a gap by providing an automated construction for integrating guess and check programs. They relieve the user from the burden to use sophisticated techniques such as saturation, as employed e.g. in [13, 8, 24], in order to overcome the technical intricacies in combining natural guess and check parts into a single program. Furthermore, our results complement recent results about meta-interpretation techniques in ASP, cf. [28, 7, 9].

The rest of this paper is organized a follows. In the next section, we very briefly recall the necessary concepts and fix notation. After that, we present in Section 3 our transformation $tr(\Pi)$ of a "checking" program Π into a disjunctive logic program. We start there with making the informal desirable properties described above more precise, present the constituents of $tr(\Pi)$, the factual program representation $F(\Pi)$ and a meta-interpreter Π_{meta}, and prove that our transformation satisfies the desirable properties. Section 4 thereafter is devoted to modifications towards optimization. In Section 5, we show how to synthesize separate guess and check programs to integrated encodings. Several applications are considered in Section 6, and experimental results for these are reported in Section 7. The final Section 8 gives a summary and presents issues for further research.

2 Preliminaries

We assume that the reader is familiar with logic programming and answer set semantics, see [15, 35], 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 "$\neg$" is the strong negation symbol, for which we also use the customary "–", in a function-free first-order language (including at least one constant), which is customarily given by the programs considered. We write $|a| = |\neg a| = a$ to denote the atom of a literal.

Extended disjunctive logic programs (EDLPs; or simply programs) are disjunctive logic programs with default (weak) and strong negation, i.e., finite sets Π of rules r

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

$l, m, n \geq 0$, where each h_i and b_j is a literal and `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 (positive, resp. negative) body of rule r. Rules with $|H(r)|{=}1$ and $B(r){=}\emptyset$ are called *facts* and rules with $H(r){=}\emptyset$ are called *constraints*. For convenience, 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.

Rules (resp. programs) are *positive*, if "`not`" does not occur in them, and *normal*, if $|H(r)| \leq 1$. A ground program Π is *head-cycle free* [3], if no literals $l \neq l'$ occurring in the same rule head mutually depend on each other by positive recursion; Π is stratified [36, 37], if no literal l depends by recursion through negation on itself (counting disjunction as positive recursion).

The *answer set semantics* [15] for DLPs is as follows. Denote by $Lit(\Pi)$ the set of all ground literals for a program Π. Consider first positive (ground) programs Π. Let $S \subseteq Lit(\Pi)$ be a set of consistent literals. Such a set S satisfies a positive rule r, if $H(r) \cap S \neq \emptyset$ whenever $B^+(r) \subseteq S$. An *answer set* for Π then is a minimal (under $\subseteq$) set S satisfying all rules.[1] To extend this definition to programs with weak negation, the *reduct* Π^S of a program Π with respect to a set of literals S is the set of rules

$$h_1 \ \mathtt{v} \ \ldots \ \mathtt{v}\ h_l \ \mathtt{:-}\ b_1,\ \ldots,\ b_m$$

for all rules (1) in Π such that $S \cap B^-(r) = \emptyset$. Then S is an *answer set* of Π, if S is an answer set for Π^S.

There is a rich literature on characterizations of answer sets of DLPs and restricted fragments; for our concerns, we recall here the following characterization of (consistent) answer sets for HDLPs, given by Ben-Eliyahu and Dechter [3]:

Theorem 1 *Given a ground HDLP Π, a consistent $S \subseteq Lit(\Pi)$ is an answer set iff*

1. *S satisfies each rule in Π, and*
2. *there is a function $\phi : Lit(\Pi) \mapsto \mathbb{N}$ such that for each literal l in S there is a rule r in Π with*
   (a) $B^+(r) \subseteq S$
   (b) $B^-(r) \cap S = \emptyset$
   (c) $l \in H(r)$
   (d) $S \cap (H(r) \setminus \{l\}) = \emptyset$
   (e) $\phi(l') < \phi(l)$ *for each* $l' \in B^+(r)$

We will use Theorem 1 as a basis for the transformation $tr(\Pi)$ in the next section.

[1] We disregard a possible inconsistent answer set, which is not of much interest for our concerns.

3 Meta-Interpreter Transformation

As discussed in the Introduction, rewriting a given check program Π_{check} to a program Π'_{check} for integration with a separate guess program Π_{guess} into a single program $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$ can be difficult in general. The problem is that the working of the answer set semantics, to be emulated in Π'_{check}, is not easy to express there.

One difficulty is that for a given answer set S of Π_{guess}, we have to test the *non-existence* of an answer set of Π_{check} with respect to S, while Π_{solve} *should have an answer set* extending S to Π'_{check} if the check succeeds. A possibility to work around this problem is to design Π'_{check} in a way such that it has a dummy answer set with respect to S if the check of Π_{check} on S succeeds, and no answer set if the check fails, i.e., if Π_{check} has some answer set on S. While this may not look to be very difficult, the following observations suggest that this is not straightforward.

Since Π_{solve} may need to solve a Σ_2^P-complete problem, any suitable program Π'_{check} must be truly disjunctive in general, i.e., contain disjunctions which are not head-cycle free (assuming that no head literal in Π'_{check} occurs in Π_{guess}). Indeed, if both Π_{guess} and Π'_{check} are head-cycle free, then also $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$ is head-cycle free, and thus can only express a problem in NP.

Furthermore, we can make in Π'_{check} only limited use of default negation on atoms which do not occur in Π_{guess}. The reason is that upon a "guess" S for an answer set of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$, the reduct Π^S_{solve} is `not`-free. Contrary to the case of Π_{check} in the two-step approach, it is not possibile to explicitly consider for a guess S_{guess} of an answer set of Π_{guess} varying extensions $S = S_{guess} \cup S'_{check}$ to the whole program Π_{solve} which activate different rules in Π'_{check} (e.g., unstratified clauses a :− not b and b :− not a encoding a choice among a and b). Therefore, default negation in rules of Π_{check} must be handled with care and might cause major rewriting as well.

These observations provide some evidence that a rule-rewriting approach for obtaining Π'_{check} from Π_{check} may be complicated. For this reason, we adopt at a generic level a Meta-interpreter approach, in which the co-NP-check modeled by Π_{check} is "emulated" by a minimality check for a positive DLP Π'_{check}.

3.1 Basic approach

The considerations above lead us to an approach in which the program Π'_{check} is constructed by the use of meta-interpretation techniques [28, 7, 9]. The idea behind meta-interpretation is here 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 the meta-interpreters available are normal logic programs (including arbitrary negation), and can not be used for our purposes for the reasons explained above. We thus have to construct a novel meta-interpreter which is essentially `not`-free, i.e. uses negation as failure only in a restricted way, and contains disjunction.

Basically, we present a general approach to translate normal LPs and HDLPs into stratified disjunctive logic programs. To this end, we exploit Theorem 1 as a basis for a transformation $tr(\Pi)$ from a given HDLP Π to a DLP $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$ such that $tr(\Pi)$ fulfills the properties mentioned in the introduction. More precisely, it will satisfy the following properties:

T0 $tr(\Pi)$ is computable in time polynomial in the size of Π.

T1 Each answer set S' of the transformed program $tr(\Pi)$ corresponds to an answer set S of Π, such that $S = \{l \mid \texttt{inS}(l) \in S'\}$ for some predicate `inS(·)`, provided Π is consistent, and conversely, each answer set S of Π corresponds to some answer set S' of $tr(\Pi)$ such that $S = \{l \mid \texttt{inS}(l) \in S'\}$.

T2 If the program Π has no answer set, 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., $tr(\Pi) = \bigcup_{r\in\Pi} tr(r)$ holds. Moreover, $tr(\Pi)$ returns a stratified DLP [36, 37] which uses negation only in its "deterministic" part.

Note that properties **T0** – **T4** for $tr(\cdot)$ are similar yet different from the notion of polynomial faithful modular (PFM) transformation by Janhunen [19, 20], which is a function Tr mapping a class of logic programs $\mathcal{C}$ to another class $\mathcal{C}'$ of logic programs (where $\mathcal{C}'$ is assumed to be a subclass or superclass of $\mathcal{C}$), such that the following three conditions hold: (1) For each program $\Pi \in \mathcal{C}$, $Tr(\Pi)$ is computable in polynomial time in the size of Π (called *polynomiality*), (2) the Herbrand base of Π, $Hb(\Pi)$, is included in the Herbrand base of $Tr(\Pi)$, $Hb(Tr(\Pi))$ and the models/interpretations of Π and $Tr(\Pi)$, are in one-to-one correspondence and coincide up to $Hb(\Pi)$ (*faithfulness*), and (3) $Tr(\Pi_1 \cup \Pi_2) = Tr(\Pi_1) \cup Tr(\Pi_2)$ for all programs Π_1, Π_1 in $\mathcal{C}$ and $\mathcal{C}' \subseteq \mathcal{C}$ implies $Tr(\Pi) = \Pi$ for all Π in $\mathcal{C}'$ (*modularity*).

Compared to PFM, also our transformation $tr(\cdot)$ is polynomially computable by **T0** and hence satisfies condition 1). Moreover, by **T4** and the fact that stratified disjunctive programs are not necessarily head-cycle free, it also satisfies condition 3). However, condition 2) fails. Its first part, that $Hb(\Pi) \subseteq Hb(tr(\Pi))$ and that answer sets coincide on $Lit(\Pi)$ could be fulfilled by adding rules l :- $\mathtt{inS}(l)$ for every $l \in Lit(\Pi)$); these polynomially many rules could be added during input generation. The second part of condition 2) is clearly in contradiction with **T2**, since for Ω never a corresponding answer set of Π exists. Moreover, condition **T1** is a weaker condition than the one-to-one correspondence between the answer sets of Π and $tr(\Pi)$ required for faithfulness: In fact, in case Π has positive cycles, there might be several possible guesses for ϕ for an answer set S of Π in Theorem 1 reflected by different answer sets of $tr(\Pi)$. We illustrate this by a short example:

Example 1 Let Π be the program consisting of the following four rules:

$r1$: `a :- b.` $r2$: `b :- a.` $r3$: `a.` $r4$: `b.`

Then, Π has a single answer set $S = \{\mathtt{a}, \mathtt{b}\}$, while $tr(\Pi)$ has two answer sets such that $S_1 = \{\mathtt{inS}(\mathtt{a}), \mathtt{inS}(\mathtt{b}), \mathtt{phi}(\mathtt{a}, \mathtt{b}), \ldots\}$ and $S_2 = \{\mathtt{inS}(\mathtt{a}), \mathtt{inS}(\mathtt{b}), \mathtt{phi}(\mathtt{b}, \mathtt{a}), \ldots\}$, intuitively reflecting that here the order of applications of rules $r1$ and $r2$ does not matter, although they are cyclic.

We remind that the different properties of our transformation $tr(\cdot)$ and PFM transformations is not an accident but a necessary feature, since we want to express nonexistence of certain answer sets via the transformation, and not merely preserve the exact semantics as targeted by PFM. Apart from this different objective, the other properties involved (polynomiality and modularity) are in effect the same.

3.2 Input representation $F(\Pi)$

As input for our meta-interpreter Π_{meta}, which will be introduced in the next subsection, we choose the representation $F(\Pi)$ of the propositional program Π defined below. We assume that each rule r has a unique name $n(r)$; for convenience, we identify r with $n(r)$.

Definition 1 *Let Π be any ground (propositional) HDLP. The set $F(\Pi)$ consists of the facts*

`lit(h,`l`,`r`).` `atom(`l`,`$|l|$`).` *for each literal* $l \in H(r)$,
`lit(p,`l`,`r`).` *for each literal* $l \in B^+(r)$,
`lit(n,`l`,`r`).` *for each literal* $l \in B^-(r)$,

for every rule $r \in \Pi$.

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.

3.3 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 `inS(·)`, is an answer set of Π by testing the criteria of Theorem 1. The steps of the transformation cast the various conditions there into rules of Π_{meta}, and also provide auxiliary machinery which is needed 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, 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.
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 the rules on lines 1-14 plus $F(\Pi)$ form a stratified program, which has a single answer set, cf. [36, 37].

Step 2 Next, 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 condition 2 of Theorem 1. We will explain this correspondence in more detail below (cf. proof of Theorem 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; only these can belong to an answer set S, but no others (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 will check whether S and `phi` violate the conditions of Theorem 1 by deriving the distinct atom `notok` (considered in Step 5 below) 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 default 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 with respect to `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 $l \in S$. This is checked by iterating over all rules with $l \in H(r)$ using the order from Step 1. Thus, condition 2.(c) is implicitly checked by this iteration.

Step 5 Whenever `notok` is derived, indicating a wrong guess, then we apply a saturation technique as in [13, 8, 24] 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`.

Definition 2 *The program* Π_{meta} *consists of the rule 1–42 from above.*

We then can formally define our transformation $tr(\Pi)$ as follows.

Definition 3 *Given any ground HDLP* Π*, its transformation* $tr(\Pi)$ *is given by the DLP* $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$.

Examples of $tr(\Pi)$ will be provided in Section 6.

3.4 Properties of $tr(\Pi)$

We now show that $tr(\Pi)$ satisfies indeed the properties **T0** – **T4** from the beginning of this section.

As for **T0**, we note the following proposition, which is not difficult to establish.

Proposition 1 *Given* Π*, the transformation* $tr(\Pi)$ *and its ground instantiation are both computable in logarithmic workspace (and thus in polynomial time).*

Proof. The input representation $F(\Pi)$ is easily generated in a linear scan of Π, using the rule numbers as names, for which a counter (representable in logspace) is sufficient. The meta-interpreter part Π_{meta} is fixed anyway. A naive grounding of $tr(\Pi)$ can be constructed by instantiating each rule r from Π_{meta} with constants from Π and rule ids in all possible ways; for each variable `X` in r, all constants of Π can be systematically considered, using counters to mark the start and end position in Π (viewed as a string), and the rule ids by a rule number counter. A constant number of such counters is sufficient. Thus, the grounding of $tr(\Pi)$ is constructible in logarithmic work space. Notice that intelligent, efficient grounding methods such as those used in DLV [23] usually generate a smaller ground program than this naive ground instantiation. $\Box$

Clearly, $tr(\Pi)$ satisfies property **T3**, and as easily checked, $tr(\Pi)$ is modular. Moreover, strong negation does not occur in $tr(\Pi)$ and weak negation only stratified. The latter is not applied to literals depending on disjunction; it thus occurs only in the deterministic part of $tr(\Pi)$, which means **T4** holds.

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

Definition 4 *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 $\Pi_\Omega = F(\Pi) \cup \bigcup_{i\in\{1,3,4,5\}} \Pi^i_{meta} \cup \{\mathtt{notok}.\}$. Then, Ω is defined as the answer set of Π_Ω.*

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

Proof. (Sketch) This follows immediately from the fact that Π_Ω is a (locally) stratified normal logic program without $\neg$ and constraints, which as well-known has a single answer set. $\Box$

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

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

Proof. 1. The first part follows immediately from the fact that $tr(\Pi)$ has no constraints, no strong negation, and weak negation is stratified; this guarantees the existence of at least one answer set S of $tr(\Pi)$ [37]. Moreover, $S' \subseteq \Omega$ must hold for every answer set: after removing $\{\mathtt{notok.}\}$ from Π_Ω and adding Π^2_{meta}, we obtain $tr(\Pi)$. Note that any rule in Π^2_{meta} fires with respect to S' only if all literals in its head are in Ω, and `inS`, `ninS`, and `phi` are elsewhere not referenced recursively through negation or disjunction. Therefore, increasing S' locally to the value of Ω on `inS`, `ninS`, `phi`, and `notok`, and closing off thus increases it globally to Ω, which means $S' \subseteq \Omega$.
2. ($\Rightarrow$) Assume that S is an answer set of Π. Clearly, then S is a consistent set of literals which has a corresponding set $S'' = \{\mathtt{inS}(l) \mid l \in S\} \cup \{\mathtt{ninS}(l) \mid l \in Lit(\Pi) \setminus S\}$ being one possible guess by the rules in lines 15 to 17 of Π_{meta}. Let now $\phi : Lit(\Pi) \rightarrow \mathbb{N}$ be the function from Theorem 1 for answer set S: Without loss of generality, we may assume two restrictions on this function ϕ:

- $\phi(l) = 0$ for all $l \in Lit(\Pi) \setminus S$ and $\phi(l) > 0$ for all $l \in S$.
- $\phi(l) \neq \phi(l')$ for all $l, l' \in S$.

Then, the function ϕ can be mapped to a total order over S `phi` such that

$$\mathtt{phi}(l,l') \Leftrightarrow \phi(l) > \phi(l') > 0.$$

This relation `phi` fixes exactly one possible guess by the lines 18 and 19 of Π_{meta}.

Note that it is sufficient to define `phi` only over literals in S: Violations of condition 2.(e) have only to be checked for rules with $B^+(r) \subseteq S$, as otherwise condition 2.(a) already fails. Obviously, condition 2.(e) of Theorem 1 is violated with respect to ϕ iff (a) `phi(Y,X)` holds for some `X` in the head of a rule with `Y` in its positive body or (b) if `X` itself occurs in its positive body. While (a) is checked in lines 36, (b) is implicit by definition of predicate `rule` (line 1) which says that a literal can not prove itself.

Given S'' and `phi` from above, we can now verify by our assumption that S is an answer set and by the conditions of Theorem 1 that (a) `notok` can never be derived in $tr(\Pi)$ and (b) S'' and `phi` uniquely determine an answer set S' of $tr(\Pi)$ of the form we want to prove. This can be argued by construction of Steps 3 and 4 of $tr(\Pi)$, where `notok` will only be derived if some rule is unsatisfied (Step 3) or there is a literal in S (i.e. S'') which fails to be proved by all other rules (Step 4).

($\Leftarrow$) Assume that S' is an answer set of $tr(\Pi)$ not containing `notok`. Then by the guess of `phi` in Step 5 a function $\phi : Lit(\Pi) \to \mathbb{N}$ can be constructed by the implied total order of `phi` as follows: We number all literals $l \in S = \{l \mid \mathtt{inS}(l) \in S'\}$ according to that order from 1 to $|S|$ and fix $\phi(l) = 0$ for all other literals. Again, by construction of Steps 3 to 5 and the assumption that `notok` $\notin S'$, we can see that S and the function ϕ constructed fulfill all the conditions of Theorem 1; in particular, line 17 guarantees consistency. Hence S is an answer set of Π.

3. ($\Leftarrow$) Assume that Π has an answer set. Then, by the already proved Part 2 of the Theorem, we know that there exists an answer set S' of $tr(\Pi)$ such that `notok` $\notin S'$. By minimality of answer sets, Ω can not be an answer set of $tr(\Pi)$.

($\Rightarrow$) By Part 1 of Theorem 2, we know that $tr(\Pi)$ always has an answer set $S' \subseteq \Omega$. Assume that there is an answer set $S' \subsetneqq \Omega$. We distinguish 2 cases: (a) `notok` $\notin S'$ and (b) `notok` $\in S'$. In case (a), proving Part 2 of this proposition, we have already shown that Π has an answer set; this is a contradiction. On the other hand, in case (b) the final "saturation" rules in Step 5 "blow up" any answer set containing `notok` to Ω, which contradicts the assumption $S' \subsetneqq \Omega$. □

As noticed above, the transformation $tr(\Pi)$ 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, such a 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 (i.e. such that $tr'(\Pi) = \bigcup_{r\in\Pi} tr'(r)$), satisfying* **T1** *such that $tr'(\Pi)$ is a positive program.*

Proof. Assuming such a transformation exists, we derive a contradiction. Let $\Pi_1 = \{$ `a :- not b.`$\}$ and $\Pi_2 = \Pi_1 \cup \{$`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 $tr'(\Pi_1)$. Since $tr'(\Pi_1)$ is a positive program, S_2 contains some answer set S_1 of $tr'\Pi_1$. By **T1**, we have that `inS(a)` $\in S_1$ must hold, and hence `inS(a)` $\in S_2$. By **T1** again, it follows that Π_2 has an answer set S such that `a` $\in S$. But the single answer set of Π_2 is $\{$`b`$\}$, which is a contradiction. □

We remark that Prop. 2 remains true if **T1** is generalized such that the answer set S of Π corresponding to S' is given by $S = \{l \mid S' \models \Psi(l)\}$, where $\Psi(x)$ is a monotone query (e.g., computed by a normal positive program without constraints). Moreover, if a successor predicate `next(X,Y)` and predicates `first(X)` and `last(X)` for the constants are available, given that the universe is finite by the constants in Π and rule names, then computing the negation of the non-input predicates accessed through **not** is feasible by a positive normal program, since such programs capture polynomial time computability by well-known results on the expressive power of Datalog [32]; thus, negation of input predicates in $F(\Pi)$ is sufficient in this case.

4 Modifications towards Optimization

The meta-interpreter Π_{meta} from above can be modified in several respects. We discuss in this section some modifications which, though not necessarily reducing the size of the ground instantiation, intuitively prune the search of an answer set solver applied to $tr(\Pi)$.

4.1 Giving up modularity (OPT$_{mod}$)

If we sacrifice modularity and allow that Π_{meta} partly depends on the input, then we can circumvent the iterations in Step 3 and in part of Step 1. Intuitively, instead of iterating over the heads and bodies of all rules in order to determine whether these rules are satisfied, we add a single rule in $tr(\Pi)$ for each rule r in Π firing **notok** whenever r is unsatisfied. We therefore replace the rules from Step 3 by

$$\begin{aligned}\mathtt{notok} \mathrel{:-} & \mathtt{ninS}(h_1),\ \ldots,\ \mathtt{ninS}(h_l),\ \mathtt{inS}(b_1),\ \ldots,\ \mathtt{inS}(b_m),\\ & \mathtt{ninS}(b_{m+1}),\ \ldots\ \mathtt{ninS}(b_n).\end{aligned} \tag{2}$$

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 become unnecessary and can be dropped.

We can even refine this further. For every normal rule $r \in \Pi$ with non-empty head, i.e. $H(r) = \{h\}$, which has a satisfied body, we can force the guess of h: we replace (2) by

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

In this context, since constraints only serve to "discard" unwanted models but cannot prove any literal, we can ignore them during input generation $F(\Pi)$. Note that dropping input representation $\mathtt{lit}(\mathtt{n}, l, c).$ 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, by removing all $l \in B^-(c)$ which do not occur in any rule head in Π. On the other hand, all literals $l \in B^-(c)$ which appear in some other (non-constraint) rule r are not critical, since facts `lit(`hpn, l, r`).` ($hpn \in \{$`h`, `p`, `n`$\}$) from this other rule will ensure that either line 15 or line 16 in Π_{meta} is applicable and therefore, either `inS(`l`)` or `ninS(`l`)` will be derived. Thus, after elimination of critical literals in constraints beforehand, we can safely drop the factual representation of constraints completely (including `lit(n,`l, c`).` for the remaining negative literals).

4.2 Restricting to potentially applicable rules (OPT$_{pa}$)

We only need to consider literals in heads of *potentially applicable* rules. These are all rules with empty bodies, and rules where any positive body literal – recursively – is the head of another potentially applicable rule. This suggests the following definition:

Definition 5 *A set R of ground rules is* potentially applicable*, if there exists an enumeration $\langle r_i \rangle_{i \in I}$ of R, where I is a prefix of $\mathbb{N}$ resp. $I{=}\mathbb{N}$, such that $B^+(r_i) \subseteq \bigcup_{j<i} H(r_j)$.*

The following proposition is then not difficult to establish.

Proposition 3 *Let* Π *be any ground HDLP. Then there exists a unique maximal set* $R^* \subseteq \Pi$ *of potentially applicable rules, denoted by* $\mathrm{PA}(\Pi)$.

Proof. Indeed, suppose $\langle r_i \rangle_{i\in I}$ and $\langle r'_i \rangle_{i\in I'}$ are enumerations witnessing that rule sets R and R' such that $R, R' \subseteq \Pi$ are potentially applicable. Then their union $R \cup R'$ is potentially applicable, witnessed by the enumeration obtained from the alternating enumeration $r_0, r'_0, r_1, r'_1, \ldots$ whose suffix are the rules from the larger set of R and R' if they have different cardinalities, from which duplicate rules are removed (i.e., remove any rule r'_j if $r'_j = r_i$, for some $i \leq j$, and remove any rule r_j if $r'_i = r_j$ and for some $i < j$). It follows that a unique largest set $R^* \subseteq \Pi$ of potentially applicable rules exists. □

The set $\mathrm{PA}(\Pi)$ can be computed by adding a rule:

```
pa(r) :- lit(h,b1,R1), pa(R1), ..., lit(h,bm,Rm), pa(Rm).
```

for any rule r of the form (1) in Π. In particular, if $m = 0$ we simply add the fact `pa(r)`. Finally, we change line 1 in Π_{meta} to:

```
rule(L,R) :- lit(h,L,R), not lit(p,L,R), not lit(n,L,R), pa(R).
```

such that only "interesting" rules are considered.

We note, however, that computing `pa(·)` incurs some cost: Informally, a profit of optimization $\mathbf{OPT}_{pa}$ might only be expected in domains where Π_{check} contains a a reasonable number of rules which positively depend on each other and might on the other hand likely be "switched off" by particular guesses in Π_{guess}.

4.3 Optimizing the order guess ($\mathbf{OPT}_{dep}$)

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 with respect to S.[2] These dependencies with respect to 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 any other rule where `phi` appears (lines 36,40) to check `phi` only in case Π has *any* cyclic dependencies with respect to S.

In the following, we will denote the transformation obtained by the optimizations from this section as $tr_{Opt}(\Pi)$ while we refer to $tr(\Pi)$ for the original transformation.

[2] Similarly, in [3] $\phi : Lit(\Pi) \rightarrow \{1, \ldots, r\}$ is only defined for a range r bound by the longest acyclic path in any strongly connected component of the program.

5 Integrating Guess and co-NP Check Programs

In this section, we show how our transformation tr (resp. tr_{Opt}) from above can be used to automatically combine a HDLP Π_{guess} which guesses in its answer sets solutions of a problem, and a HDLP Π_{check} which encodes a co-NP-check of the solution property, into a single DLP Π_{solve} of the form $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$.

We assume that the set $Lit(\Pi_{guess})$ is a Splitting Set [25] for $\Pi_{guess} \cup \Pi_{check}$, i.e. no head literal from Π_{check} occurs in Π_{guess}. This can be easily achieved by introducing new predicate names, e.g., $\mathtt{p}'$ for a predicate $\mathtt{p}$, and adding a rule $\mathtt{p}'(t)\,\texttt{:-}\,\mathtt{p}(t)$ in case there is an overlap.

Each rule r in Π_{check} is of the form

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

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

Definition 6 *For any ground program Π_{check} as above, the program Π'_{check} contains the following rules and constraints:*

(i) The facts $F(\Pi_{check})$ in a conditional version: For each rule $r \in \Pi_{check}$ of form (4), the rules

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

(ii) each rule in $\Pi_{meta} = tr(\Pi_{check}) \setminus F(\Pi_{check})$ (resp. in $tr_{Opt}(\Pi_{check}) \setminus F(\Pi_{check})$, where $\mathsf{body}_{guess}(r)$ must be added to the bodies of the rules (2) and (3));

(iii) a constraint

```
:- not notok.
```

It eliminates any answer set 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 *Given separate guess and check programs Π_{guess} and Π_{check}, the answer sets of*

$$\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check},$$

denoted S_{solve}, are in 1-1 correspondence with the answer sets S of Π_{guess} such that $\Pi_{check} \cup S$ has no answer set.

Proof. This result can be derived from Theorem 2 and the Splitting Set Theorem for logic programs under answer set semantics [25]. We consider the proof for the original transformation $tr(\cdot)$; the proof for the optimized transformation $tr_{Opt}(\cdot)$ is similar (with suitable extensions in places). In what follows, for any program Q and any consistent literal set S, we let $Q[S]$ denote the program obtained from Q by eliminating every rule r such that $\mathsf{body}_{guess}(r)$ is false in S, and by removing $\mathsf{body}_{guess}(r)$ from the remaining rules. Notice that $\Pi_{check} \cup S$ and $\Pi_{check}[S] \cup S$ have the same answer sets.

We can rewrite Π_{solve} as

$$\Pi_{solve} = \Pi_{guess} \cup F'(\Pi_{check}) \cup \Pi_{meta} \cup \{\ \texttt{:- not notok.}\ \}$$

where $F'(\Pi_{check})$ denotes the modified factual representation for Π_{check}, given in item 1. of the definition of Π'_{check}. By hypothesis on $\Pi_{guess} \cup \Pi_{check}$, the set $Lit(\Pi_{guess})$ is a splitting set for Π_{solve}. Hence, as easily seen also $Lit(\Pi_{guess} \cup F'(\Pi_{check}))$ is a splitting set for Π_{solve}, and $Lit(\Pi_{guess})$ is also a splitting set for $\Pi_{guess} \cup F'(\Pi_{check})$. Moreover, each answer set S of Π_{guess} is in 1-1 correspondence with an answer set S' of $\Pi_{guess} \cup F'(\Pi_{check})$. Then $S' \setminus S = F(\Pi_{check}[S]) \cup A_S$, such that $F(\Pi_{check}[S])$ is the factual representation of $\Pi_{check}[S]$ in the transformation $tr(\Pi_{check}[S])$ and $A_S = \{\texttt{atom}(l, |l|).\mid l \in H(\Pi_{check}) \setminus H(\Pi_{check}[S])\}$[3] is an additional set of facts emerging from $F'(\Pi_{check})$, since we added facts $\texttt{atom}(l, |l|).$ for all head literals of $r \in \Pi_{check}$, not only for those r where $\mathsf{body}_{guess}(r)$ was satisfied.

Now let S_{solve} be any (consistent) answer set of Π_{solve}. From the Splitting Set Theorem [25], we can conclude that S_{solve} can be written as $S_{solve} = S \cup S_{check} \cup A_S$ where S and $S_{check} \cup A_S$ are disjoint, S is an answer set of Π_{guess}, and $S_{check} \cup A_S$ is an answer set of the program $\Pi'_S = (\Pi_{solve} \setminus \Pi_{guess})[S]$. Since $F'(\Pi_{check})$ is the only part of $\Pi_{solve} \setminus \Pi_{guess}$ where literals from $Lit(\Pi_{guess})$ occur, we obtain

$$\begin{aligned} \Pi'_S &= F(\Pi_{check}[S]) \cup A_S \cup \Pi_{meta} \cup \{\texttt{:- not notok.}\} \\ &= tr(\Pi_{check}[S]) \cup A_S \cup \{\texttt{:- not notok.}\}. \end{aligned}$$

The additional facts A_S can be viewed as independent part of any answer set of Π'_S, since the answer sets of Π'_S are the sets $T \cup A_S$ where T is any answer set of $\Pi'_S \setminus A_S$; note that $T \cap A_S = \emptyset$. Indeed, the only rule in Π'_S where the facts of A_S play a role, is line 17 of Π_{meta}. All ground instances of line 17 are of the following form:

```
notok :- inS(l), inS(nl), l!=nl, atom(l,|l|), atom(nl,|l|).
```

We assume r fires and `atom(l,|l|)` $\in A_S$ (resp. `atom(nl,|l|` $\in A_S$). Then, in order for the rule to fire, `inS(l)` (resp. `inS(nl)`) has to be true. However, this can only be the case for literals `l` (resp. `nl`) occurring in a rule head of $\Pi_{check}[S]$ (backwards, by the rules in line 15, 14 and 1 of Π_{meta} and by definition of Π'_{check}), which contradicts our assumption that `atom(l,|l|)` $\in A_S$ (resp. `atom(nl,|l|` $\in A_S$). Therefore, the facts of A_S do not affect the rule in line 17 and consequently Π'_S has an answer set if and only if $\Pi'_S \setminus A_S$ has an answer set and these answer sets coincide on $Lit(\Pi'_S) \setminus A_S$.

By Theorem 2, we know that (i) $tr(\Pi_{check}[S])$ always has an answer set and (ii) $tr(\Pi_{check}[S])$ has any answer set containing `notok` (which is unique) if and only if $\Pi_{check}[S]$ has no answer set. However, the constraint `:- not notok.` only allows for answer sets of Π'_S containing `notok`. Hence, an answer set S_{check} of $\Pi'_S \setminus A_S$ exists if and only if $\Pi_{check}[S]$ has no answer set, equivalently, $\Pi_{check} \cup S$ has no answer set.

[3]Here, for any program Π, we write $H(\Pi) = \bigcup_{r \in \Pi} H(r)$.

Conversely, suppose S is an answer set of Π_{guess} such that $\Pi_{check} \cup S$ has no answer set; equivalently, $\Pi_{check}[S]$ has no answer set. By Theorem 2, we know that $tr(\Pi_{check}[S]) = F(\Pi_{check}[S]) \cup \Pi_{meta}$ has a unique answer set S_{check}, and S_{check} contains `notok`. Hence, also the program

$$Q_S = F(\Pi_{check}[S]) \cup \Pi_{meta} \cup \{\texttt{:- not notok.}\}$$

has the unique answer set S_{check}. On the other hand, since S is an answer set of Π_{guess} and $Lit(\Pi_{guess})$ is a splitting set for Π_{solve}, for each answer set S'' of the program $\Pi'_S = (\Pi_{solve} \setminus \Pi_{guess})[S]$, we have that $S \cup S''$ is an answer set of Π_{solve}. However, $\Pi'_S = Q_S \cup A_S$; hence, $S'' = S_{check} \cup A_S$ must hold and $S_{solve} = S \cup S_{check} \cup A_S$ is the unique answer set of Π_{solve} which extends S. This proves the result. □

The optimizations $\mathbf{OPT}_{pa}$ and $\mathbf{OPT}_{dep}$ in Section 4 still apply. However, concerning $\mathbf{OPT}_{mod}$, the following modifications are necessary:

1. Like the input representation, rules (2) and (3) have to be extended by adding body_{guess}`(r)`.

2. As for constraints c, we mentioned above that the factual representation of literals in $B(c)$ may be skipped. This now only applies to literals in $B^+(c)$; the rule `lit(n,`l,c`) :-` $\mathsf{body}_{guess}(c)$`.` for $l \in B^-(c)$ may no longer be dropped in general, as shown by the following example.

Example 2 Let $\Pi_{guess} = \{$ `g v-g.` $\}$ and $\Pi_{check} = \{\, r1:$ `x :- g.`, $r2:$ `:- not x.` $\}$ The "input" representation of Π_{check} with respect to optimization $\mathbf{OPT}_{mod}$, i.e., the variable part of Π'_{check}, now consists of:

`lit(h,x,`$r1$`) :-g.   lit(n,x,`$r2$`).   inS(x) :-g.   notok :-ninS(x).`

where the latter correspond to rules (3) and (2). If we now assume that we want to check answer set $S = \{$`-g`$\}$ of Π_{guess}, it is easy to see that Π_{check} has no answer set for S, and therefore S should be represented by some answer set of our integrated encoding. Now assume that `lit(n,x,`$r2$`).` is dropped and we proceed in generating the integrated encoding as outlined above with respect to $\mathbf{OPT}_{mod}$. Since `g` $\notin S$ and we have dropped `lit(n,x,`$r2$`).`, the "input" representation of Π_{check} for S comprises only the final rule `notok :- ninS(x).`. However, this rule can never fire because neither line 15 nor line 16 of Π_{meta} can ever derive `ninS(c)`. Therefore, also `notok` can not be derived and the integrated check fails. On the other hand, `lit(n,x,`$r2$`).` suffices to derive `ninS(x)` from line 16 of Π_{meta}, such that `notok` can be derived and the integrated check works as intended.

In certain cases, we can still drop $l \in B^-(c)$. For example, if l occurs in the head of a rule r with $\mathsf{body}_{guess}(r) = \emptyset$, since in this case `lit(h,l,`r`)` will always be added to the program (see also respective remarks in Section 6).

5.1 Integrating Guess and NP Check Programs

In contrast to the situation above, integrating a guess program Π_{guess} and a check program Π_{check} which succeeds iff $\Pi_{check} \cup S$ has *some* answer set, is easy. Given that Π_{check} is a HDLP again, this amounts to integrating a check which is in NP. After a rewriting to ensure 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.

6 Applications

We now give examples of the use of our transformation for three well-known Σ_2^P-complete problems from the literature, which involve co-NP-complete checking for a polynomial-time solution guess: the first is about quantified Boolean formulas (QBFs) with one quantifier alternation, which are well-studied in Answer Set Programming, the second about conformant planning [10, 40, 24], and the third is about strategic companies in the business domain [23].

Further examples and ad hoc encodings of such problems can be found e.g. in [13, 12, 23] (and solved similarly). However, note that our method is applicable to *any* checks encoded by inconsistency of a HDLP; co-NP-hardness is not a prerequisite.

6.1 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 $c_i = a_{i,1} \wedge \cdots \wedge a_{i,l_i}$ and $|a_{i,j}| \in \{x_1, \ldots, x_m, y_1 \ldots, y_n\}$, the problem is to compute some resp. all 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_{guess}) Guess a truth assignment for the variables $x_1, \ldots, x_m$.

(QBF_{check}) Check whether this (fixed) assignment satisfies Φ for all assignments of variables $y_1, \ldots, y_n$.

Both parts can be encoded by very simple HDLPs (or similarly by normal programs):

QBF_{guess} :

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

QBF_{check} :

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

:- $a_{1,1}, \ldots, a_{1,l_1}$.

⋮

:- $a_{k,1}, \ldots, a_{k,l_1}$.

Clearly, both programs are head-cycle free. Moreover, for every answer set S of QBF_{guess} – representing an assignment to $x_1, \ldots, x_m$– the program $QBF_{check} \cup S$ has no answer set thanks to the constraints, iff every assignment for $y_1, \ldots, y_n$ satisfies formula Φ.

By the method described in Section 5, we can automatically generate a single program Π_{solve} integrating the guess and check programs. For illustration, we consider the following QBF:

$$\exists x_0 x_1 \forall y_0 y_1 (\neg x_0 \wedge \neg y_0) \vee (y_0 \wedge \neg x_0) \vee (y_1 \wedge x_0 \wedge \neg y_0) \vee (y_0 \wedge \neg x_1 \wedge \neg y_0)$$

This QBF evaluates to true: for the assignments $x_0 = 0, x_1 = 0$ and $x_0 = 0, x_1 = 1$, the subformula $\forall y_0 y_1 (\cdots)$ is a tautology.

The integrated program $QBF_{solve} = QBF_{guess} \cup QBF'_{check}$ under use of the optimized transformation $tr_{Opt}(\cdot)$ of $tr(\cdot)$ as discussed is shown in Figure 1. It has two answer sets of the form $S_1 = \{x_0, -x_1, \ldots, \}$ and $S_2 = \{x_0, x_1, \ldots, \}$, respectively.

Figure 1: Integrated encoding QBF_{solve} for QBF $\exists x_0 x_1 \forall y_0 y_1 (\neg x_0 \wedge \neg y_0) \vee (y_0 \wedge \neg x_0) \vee (y_1 \wedge x_0 \wedge \neg y_0) \vee (y_0 \wedge \neg x_1 \wedge \neg y_0)$

```
%%%% GUESS PART
 x0 v -x0.  x1 v -x1.

%%%% REWRITTEN CHECK PART
%% 1. Create dynamically the facts for the check program:

% y0 v -y0.
  lit(h,"y0",1). lit(h,"-y0",1).
  atom("y0","y0"). atom("-y0","y0").

% y1 v -y1.
  lit(h,"y1",2). lit(h,"-y1",2).
  atom("y1","y1"). atom("-y1","y1").

 % :- -y0, -x0.
 % :-  y0, -x0.
 % :- -y0, y1, x0.
 % :- -y0, y0, -x1.

%% 2. Optimized meta-interpreter
%% 2.1 -- program dependent part

  notok :- ninS("y0"),ninS("-y0").
  notok :- ninS("y1"),ninS("-y1").
  notok :- inS("-y0"),-x0.
  notok :- inS("y0"),-x0.
  notok :- inS("y1"),inS("-y0"),x0.
  notok :- inS("y0"),inS("-y0"),-x1.

%% 2.2 -- fixed rules

% Iterate only over rules which contain L in the head:
  rule(L,R) :- lit(h,L,R), not lit(p,L,R), not lit(n,L,R).
  ruleBefore(L,R) :- rule(L,R), rule(L,R1), R1<R.
  ruleAfter(L,R) :- rule(L,R), rule(L,R1), R<R1.
  ruleBetween(L,R1,R2) :- rule(L,R1), rule(L,R2), rule(L,R3), R1<R3, R3<R2.
  firstRule(L,R) :- rule(L,R), not ruleBefore(L,R).
  lastRule(L,R) :- rule(L,R), not ruleAfter(L,R).
  nextRule(L,R1,R2) :- rule(L,R1), rule(L,R2), R1<R2, not ruleBetween(L,R1,R2).

% hlits are only those from active rules:
  hlit(L) :- rule(L,R).
  inS(L) v ninS(L) :- hlit(L).
  ninS(L) :- lit(HPN,L,R), not hlit(L).

% Consistency check could be skipped for programs without class. negation:
  notok :- inS(L), inS(NL), L != NL, atom(L,A), atom(NL,A).

  dep(L,L1) :- rule(L,R),lit(p,L1,R),inS(L1), inS(L).
  dep(L,L2) :- rule(L,R),lit(p,L1,R),dep(L1,L2),inS(L).
  cyclic :- dep(L,L1), dep(L1,L).
  phi(L,L1) v phi(L1,L) :- dep(L,L1), dep(L1,L), L<L1, cyclic.
  phi(L,L2) :- phi(L,L1),phi(L1,L2), cyclic.
  failsToProve(L,R) :- rule(L,R), lit(p,L1,R), ninS(L1).
  failsToProve(L,R) :- rule(L,R), lit(n,L1,R), inS(L1).
  failsToProve(L,R) :- rule(L,R), rule(L1,R), inS(L1), L1!=L.
  failsToProve(L,R) :- lit(p,L1,R), rule(L,R), phi(L1,L), cyclic.
  allFailUpto(L,R) :- failsToProve(L,R), firstRule(L,R).
  allFailUpto(L,R1) :- failsToProve(L,R1), allFailUpto(L,R), nextRule(L,R,R1).
  notok :- allFailUpto(L,R), lastRule(L,R), inS(L).
  phi(L,L1) :- notok, hlit(L), hlit(L1), cyclic.
  inS(L) :- notok, hlit(L).
  ninS(L) :- notok, hlit(L).

%%% 3. constraint
   :- not notok.
```

With respect to the variants of the transformation, we remark that for the QBF encoding considerations upon negative literals in constraints in $\mathbf{OPT}_{mod}$ do not play a role, because all literals in the constraints of QBF_{check} are positive. Also $\mathbf{OPT}_{pa}$ does not play a role, since the only rules in QBF_{check} with non-empty heads are always potentially applicable because their bodies are empty.

Note that the customary (but tricky) saturation technique in disjunctive logic programming to solve this problem, as used e.g. in [13, 23] and shown in C, is fully transparent to the non-expert, who might easily come up with the two programs above.

6.2 Conformant planning

Loosely speaking, planning is the problem of finding 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* [17] is concerned with finding a plan P which works under all contingencies which may arise because of incomplete information about the initial state and/or nondeterministic action effects.

As well-known, conformant planning in a STRIPS-style formulation is a Σ_2^P-complete problem (precisely, deciding plan existence) in certain settings, e.g. if the plan length n (of polynomial size) is given and executability of actions is guaranteed, cf. [10, 40]. 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 [30] as in [10]: 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 $Bomb_{guess}$ and $Bomb_{check}$, respectively, we can compute a plan for having the bomb disarmed by two actions:

$Bomb_{guess}$:

```
% 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).
```

$Bomb_{check}$:

```
% Initial state:
armed(0) v -armed(0).
% Frame Axioms:
armed(T1) :- armed(T), not -armed(T1), time(T), 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 whether goal holds in stage 2:
:- not armed(2).
```

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

By using the method from Section 5, the programs $Bomb_{guess}$ and $Bomb_{check}$ can be integrated automatically into a single program $Bomb_{plan} = Bomb_{guess} \cup Bomb_{check}{}'$ (cf. A). It has a single answer set, corresponding to the single conformant plan P = `dunk, flush` as desired.

We point out that our rewriting method is more generally applicable than the encoding for conformant planning proposed in [24]. It loosens some of the restrictions there: While [24] requires that the state transition function is specified by a positive constraint-free logic program, our method can still safely be used in presence of negation and constraints, provided action execution will always lead to a consistent successor state and not entail absurdity; see [10, 40] for a discussion of this setting.

Concerning $\mathbf{OPT}_{mod}$, we point out that there is the interesting constraint

c: `:- not armed(2).`

in program $Bomb_{check}$. Here, we may drop `lit(h,"armed(2)",c)` safely: For the frame axiom

r: `armed(2) :- armed(1), not -armed(2), time(1).`

(cf. A), we have $\mathsf{body}_{guess}(r) = \{$`time(1)`$\}$. Therefore, we obtain:

```
lit(h,"armed(2)",r) :- time(1).
```

However, this rule will always be added since `time(1)` is a deterministic consequence of $Bomb_{guess}$. As for $\mathbf{OPT}_{pa}$ and considering the "Bomb in the Toilet" instances from [10], there might be rules which are *not* possible applicable with respect to a guessed plan; however, in experiments, the additional overhead for computing unfounded sets did not pay off.

A generalization of the method demonstrated here on a small planning problem expressed in Answer Set Programming to conformant planning in the DLV$^{\mathcal{K}}$ planning system [10], is discussed in detail in [34]. In this system, planning problems are encoded in a logical action language, and the encodings are mapped to logic programs. For conformant planning problems, separate guess and check programs have been devised [10], which by our method can be automatically integrated into a single logic program. Such an encoding was previously unkown.

6.3 Strategic Companies

Another Σ_2^P-complete problem is the strategic companies problem from [4]. Briefly, a holding owns companies, each of which produces some goods. Moreover, several companies may jointly have control over another company. Now, some companies should be sold, under the constraint that all goods can be still produced, and that no company is sold which would still be controlled by the holding after the transaction.

PRODUCT	COMPANY #1	COMPANY #2
Pasta	Barilla	Saiwa
Tomatoes	Frutto	Barilla
Wine	Barilla	–
Bread	Saiwa	Panino

Table 1: Relation $prod\_by$ storing producers of each good

A company is *strategic*, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints. Guessing a strategic set, and checking its minimality can be done by the following two programs, where we adopt the constraint in [4] that each product is produced by at most two companies and each company is jointly controlled by at most three other companies.

SC_{guess} :

```
strat(X) v -strat(X) :- company(X).
         :- prod_by(X,Y,Z), not strat(Y), not strat(Z).
         :- contr_by(W,X,Y,Z), not strat(W),
            strat(X), strat(Y), strat(Z).
```

SC_{check} :

```
strat1(X) v -strat1(X) :- strat(X).
          :- prod_by(X,Y,Z), not strat1(Y), not strat1(Z).
          :- contr_by(W,X,Y,Z), not strat1(W),
             strat1(X), strat1(Y), strat1(Z).
 smaller  :- -strat1(X).
          :- not smaller.
```

Here, strat(C) means that C is strategic, prod_by($P, C1, C2$) that product P is produced by companies $C1$ and $C2$, and contr_by($C, C1$, $C2, C3$) that C is jointly controlled by $C1, C2$ and $C3$. We assume facts company(·)., prod_by(·,·,·)., and contr_by(·,·,·,·). to be defined in a separate program which can be considered as part of SC_{guess}.

The two programs above intuitively encode guessing a set strat of companies which fulfills the production and control preserving constraints, such that no real subset strat1 fulfills these constraints. While the ad hoc encodings from [8, 23], which can also be found in D, are not immediate (and require some thought), the above programs are very natural and easy to come up with.

As an example, let us consider the following production and control relations from [4] in a holding as shown in Tables 1 and 2. The symbol "–" there means that the entry is void, which we simply represent by duplicating the single producer (or one of the controlling companies, respectively) in the factual representation; a possible representation is thus

```
company(barilla). company(saiwa).
company(frutto).  company(panino).
prod_by(pasta,barilla,saiwa).  prod_by(tomatoes,frutto,barilla).
prod_by(wine,barilla,barilla). prod_by(bread,saiwa,panino).
contr_by(frutto,barilla,saiwa,saiwa).
```

If we would consider only the production relation, then Barilla and Saiwa together would form a strategic set, because they jointly produce all goods but neither of them alone. On the other hand, Frutto would not

CONTROLLED	CONT #1	CONT #2	CONT #3
Frutto	Barilla	Saiwa	–

Table 2: Relation $contr\_by$ storing company control information

be strategic. However, given the company control as in Table 2 means that Barilla and Saiwa together have control over Frutto. Taking into account that therefore Frutto can be sold only if either Barilla or Saiwa is also sold, the minimal sets of companies that produce all goods change completely: {Barilla, Saiwa} is no longer a strategic set, while s_1 = {Barilla, Saiwa, Frutto} is. Alternatively, s_2 = {Barilla, Panino} is another strategic set.

Integration of the programs SC_{guess} and SC_{check} after grounding is again possible by the method from Section 5 in an automatic way. Here, the facts representing the example instance are to be added as part of SC_{guess}, yielding two answer sets corresponding to s_1 and s_2 (cf. B).

With regard to **OPT**$_{mod}$, we remark that depending on the concrete problem instance, SC_{check} contains critical constraints c, where `not strat1(·)` occurs, such that `lit(n,"strat1(·)",c)` may not be dropped here (cf. B). Furthermore, as for **OPT**$_{pa}$ all rules with non-empty heads are either possibly applicable or "switched off" by SC_{guess}. Since there are no positive dependencies among the rules, `pa(·)` does not play a role there.

As a final remark, we note that modifying the guess and check programs SC_{guess} and SC_{check} to allow for unbounded numbers of producers for each product and controllers for each company, respectively, is easy. Assume that production and control are represented instead of relations `prod_by` and `contr_by` by an arbitrary number of facts of the form `produces(`c`,`p`).` and `controls(`c_1`,`g`,`c`).`, which state that company c produces p and that company c_1 belongs to a group g of companies which jointly control c, respectively. Then, we would simply have to change the constraints in SC_{guess} to:

```
no_control(G,C) :- controls(C1,G,C), not strat(C1).
:- controls(C1,G,C), not no_control(G,C), not strat(C).
produced(P) :- produces(C,P), strat(C).
:- produces(C,P), not produced(P).
```

The constraints in SC_{check} are changed similarly. Then, the synthesized integrated encoding according to our method gives us a DLP solving this problem. The ad hoc encodings in [8, 23] can not be adapted that easily, and in fact require substantial changes.

7 Experiments

As for evaluation of the proposed approach we have conducted a series of experiments for the problems outlined in the previous Section. Here, we were mainly interested in the following questions:

(1) *What is the performance impact of our automatically generated, integrated encoding compared with ad hoc encodings of* Σ_2^P *problems?*

We have therefore compared our automatically generated integrated encoding of QBFs and Strategic Companies against the following ad hoc encodings:

(i) QBF against the ad hoc encoding for QBFs described in [23] (which assumes that the quantifier-free part is in 3DNF, i.e., contains three literals per disjunct); see C.

(ii) Strategic companies against the two ad hoc encodings for the Strategic Companies problem from [8]; see D.

These two encodings significantly differ: The first encoding, $adhoc_1$ is very concise, and integrates guessing and checking in only two rules; it is an illustrative example of the power of disjunctive rules and tailored for a DLP system under answer set semantics. The second encoding, $adhoc_2$, has a more obvious separate structure of the guessing and checking parts of the problem at the cost of some extra rules. However, in our opinion, none of these ad hoc encodings is obvious at first sight compared with the separate guess and check programs shown above.

Concerning (i) we have tested randomly generated QBF instances with n existentially and n universally quantified variables (QBF-n), and concerning (ii) we have chosen randomly generated instances involving n companies (SC-n).

(2) *What is the performance impact of the automatically generated, integrated encoding compared with interleaved computation of guess and check programs?*

To this end, we have tested the performance of solving some conformant planning problems with integrated encodings compared with the ASP based planning system $\text{DLV}^{\mathcal{K}}$ [10] which solves conformant planning problems by interleaving the guess of a plan with checking plan security. For its interleaved computation, $\text{DLV}^{\mathcal{K}}$ hinges on translations of the planning problem to HDLPs, by computing "optimistic" plans as solutions of a HDLP Π^{plan}_{guess} and interleaved checking of plan security by non-existence of solutions of a new program Π^{plan}_{guess} which is dynamically generated with respect to the plan at hand. $\text{DLV}^{\mathcal{K}}$ generalizes in some sense solving the small planning example in Section 6.2 for arbitrary planning problems specified in a declarative language, $\mathcal{K}$ [11]. For our experiments we have used elaborations of "Bomb in the Toilet" as described in [11], namely "Bomb in the Toilet with clogging" BTC(i), where the toilet is clogged after dunking a package, and "Bomb in the Toilet with Uncertain Clogging" BTUC(i) where this clogging effect is non-deterministic and there are i many possibly armed packages.

7.1 Test Environment and General Setting

All tests were performed on an AMD Athlon 1200MHz machine with 256MB of main memory running SuSE Linux 8.1.

All our experiments have been conducted using the DLV system [23, 14], which is a state-of-the-art Answer Set Programming engine capable of solving DLPs. Another available system, GNT [21][4] which is not reported here showed worse performance/higher memory consumption on the tested instances.

Since our method works on ground programs, we had to ground all instances (i.e. the corresponding guess and check programs) beforehand whenever dealing with non-ground programs. Here, we have used DLV grounding with most optimizations turned off:[5] Some optimizations during DLV grounding rewrite the program, adding new predicate symbols, etc. which we turned off in order to obtain correct input for the meta-interpreters.

[4]GNT, available from `http://www.tcs.hut.fi/Software/gnt/`, is an extension of SMODELS solving DLPs by interleaved calls of SMODELS, which itself is only capable of solving normal LPs.

[5]Respective ground instances have been produced with the command `dlv -OR- -instantiate`, (cf. the DLV-Manual [14]), which turns off most of the grounding optimizations.

In order to assess the effect of various optimizations and improvements to the transformation $tr(\cdot)$, we have also conducted the above experiments with the integrated encodings based on different optimized versions of $tr(\cdot)$.

7.2 Results

The results of our experiments are shown in Tables 3-5. We report there the following tests on the various instances:

- $meta$ indicates the unoptimized meta-interpreter Π_{meta}
- mod indicates the non-modular optimization $\mathbf{OPT}_{mod}$ including the refinement for constraints.
- dep indicates the optimization $\mathbf{OPT}_{dep}$ where `phi` is only guessed for literals mutually depending on each other through positive recursion.
- *opt* indicates both optimizations $\mathbf{OPT}_{mod}$ and $\mathbf{OPT}_{dep}$ turned on.

We did not include optimization $\mathbf{OPT}_{pa}$ in our experiments, since the additional overhead for computing unfounded rules in the check programs which we have considered did not pay off (in fact, $\mathbf{OPT}_{pa}$ is irrelevant for QBF and Strategic Companies).

All times reported in the tables represent the execution times for finding the first answer set under the following resource constraints. We set a time limit of 10 minutes (=600 seconds) for QBFs and Strategic Companies, and of 4.000 seconds for the "Bomb in the Toilet" instances. Furthermore, the limit on memory consumption was 256 MB (in order to avoid swapping). A dash '-' in the tables indicates that one or more instances exceeded these limits.

The results in Tables 4-5 show that the "guess and saturate" strategy in our approach benefits a lot from optimizations for all problems considered. However, we emphasize that it might depend on the structure of Π_{guess} and Π_{check} which optimizations are beneficial. We strongly believe that there is room for further improvements both on the translation and for the underlying DLV engine.

We note the following observations:

- Interestingly, for the QBF problem, the performance of our optimized translation stays within reach of the ad hoc encoding in [23] for small instances. Overall, the performance shown in Table 3 is within roughly a factor of 5-6 (with few exceptions for small instances), and thus scales similarly.
- For the Strategic Companies problem, the picture in Table 4 is even more interesting. Unsurprisingly, the automatically generated encoding is inferior to the succinct ad hoc encoding $adhoc_1$; it is more than an order of magnitude slower and scales worse. However, while it is slower by a small factor than the ad hoc encoding $adhoc_2$ (which is more involved) on small instances, it scales much better and quickly outperforms this encoding.
- For the planning problems, the integrated encodings tested still stay behind the interleaved calls of $\text{DLV}^{\mathcal{K}}$.
- In all cases, the time limit was exceeded (for smaller instances) rather than the memory limit, but especially for bigger instances of "Bomb in the toilet" and "Strategic Companies," in some cases the memory limit was exceeded before timeout (e.g. for BTUC(5), even with the optimized version of our transformation).

	adhoc [23]		*meta*		*mod*		*dep*		*opt*	
	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
QBF-4	0.01s	0.02s	0.16s	0.18s	0.10s	0.15s	0.09s	0.11s	0.07s	0.09s
QBF-6	0.01s	0.02s	1.11s	1.40s	0.25s	1.12s	0.17s	0.21s	0.08s	0.12s
QBF-8	0.01s	0.06s	10.4s	16.3s	1.18s	7.99s	0.49s	0.87s	0.10s	0.23s
QBF-10	0.02s	0.09s	82.7s	165s	4.34s	30.7s	1.74s	3.67s	0.12s	0.36s
QBF-12	0.02s	0.16s	-	-	-	-	-	-	0.15s	0.79s
QBF-14	0.06s	1.21s	-	-	-	-	-	-	0.34s	5.87s
QBF-16	0.08s	1.85s	-	-	-	-	-	-	0.44s	10.3s
QBF-18	0.19s	7.12s	-	-	-	-	-	-	1.04s	38.8s
QBF-20	1.49s	21.3s	-	-	-	-	-	-	7.14s	101s

Average and maximum times for 50 randomly chosen instances per size.

Table 3: Experiments for QBF

	$adhoc_1$ [8]		$adhoc_2$ [8]		*meta*		*mod*		*dep*		*opt*	
	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
SC-10	0.01s	0.02s	0.05s	0.05s	0.66s	0.69s	0.49s	0.51s	0.36s	0.38s	0.13s	0.15s
SC-15	0.01s	0.02s	0.11s	0.13s	1.82s	3.23s	1.50s	3.12s	0.64s	0.68s	0.20s	0.22s
SC-20	0.02s	0.02s	0.26	0.27s	3.75s	3.90s	3.34s	3.61s	1.07s	1.13s	0.26s	0.27s
SC-25	0.02s	0.02s	0.51s	0.54s	-	-	-	-	1.63s	1.68s	0.33s	0.35s
SC-30	0.02s	0.03s	0.91s	0.97s	-	-	-	-	2.35s	2.47s	0.42s	0.44s
SC-35	0.02s	0.03s	1.50s	1.60s	-	-	-	-	3.17s	3.27s	0.54s	0.56s
SC-40	0.03s	0.03s	2.52s	2.70s	-	-	-	-	4.25s	4.43s	0.68s	0.71s
SC-45	0.03s	0.04s	4.503	4.97s	-	-	-	-	5.46s	5.77s	0.84s	0.90s
SC-50	0.03s	0.04s	8.38s	8.68s	-	-	-	-	6.73s	6.86s	1.00s	1.02s
SC-60	0.04s	0.05s	22.6s	24.3s	-	-	-	-	10.2s	10.6s	1.47s	1.53s
SC-70	0.04s	0.05s	44.2s	48.1s	-	-	-	-	14.7s	15.4s	2.05s	2.10s
SC-80	0.04s	0.05s	75.9s	82.5s	-	-	-	-	19.7s	21.0s	2.78s	3.05s
SC-90	0.05s	0.06s	125s	130s	-	-	-	-	26.8s	27.6s	3.67s	3.85s
SC-100	0.06s	0.08s	196s	208s	-	-	-	-	34.8s	36.3s	4.70s	4.80s

Average and maximum times for 10 randomly chosen instances per size.

Table 4: Experiments for Strategic Companies

	$\text{DLV}^{\mathcal{K}}$[10]	$meta$	mod	dep	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	-	-	28.1s	13.0s
BTUC(4)	577s	-	-	-	2322s

BTC, BTUC with 2,3 and 4 packages.

Table 5: Experiments for Bomb in Toilet

8 Summary and Conclusion

We have considered the problem of integrating separate "guess" and "check" programs for solving expressive problems in the Answer Set Programming paradigm with a 2-step approach, into a single logic program. To this end, we have first presented a polynomial-time transformation of a head-cycle free, disjunctive program Π into a disjunctive program $tr(\Pi)$ which is stratified and constraint-free, such that in the case where Π is inconsistent (i.e., has no answer set), $tr(\Pi)$ has a single designated answer set which is easy to recognize, and otherwise the answer sets of Π are encoded in the answer sets of $tr(\Pi)$. We then showed how to exploit $tr(\Pi)$ for combining a "guess" program Π_{solve} and a "check" program Π_{check} for solving a problem in Answer Set Programming automatically into a single disjunctive logic program, such that its answer sets encode the solutions of the problem.

Experiments have shown that such a synthesized encoding has weaker performance than the two-step method or an optimal ad hoc encoding for a problem, but can also outperform (reasonably looking) ad hoc encodings. This is noticeable since in some cases, finding any arbitrary "natural" (not necessarily optimal) encoding of a problem in a single logic program appears to be very difficult, such as e.g., for conformant planning [24] or determining minimal update answer sets [12], where such encodings were not known for the general case.

Several issues remain for being tackled in future work. The first issue concerns extending the scope of programs which can be handled. The rewriting method which we have presented here applies to propositional programs only. Thus, before transformation, the program should be instantiated. In [23] instantiations of a logic program used in DLV have been described, which keep the grounding small and do not necessarily ground over the whole Herbrand universe. For wider applicability and better scalability of the approach, a more efficient lifting of our method to non-ground programs is needed. Furthermore, improvements to the current transformations might be researched. Some preliminary experimental results suggest that a structural analysis of the given guess and check program might be valuable for this purpose.

A further issue are alternative transformations, which are possibly tailored for certain classes of programs. The work of Ben-Eliyahu and Dechter [3], on which we build, aimed at transforming head-cycle free disjunctive logic programs into SAT problems. It might be interesting to investigate whether related methods such as the one developed for ASSAT [27], which was recently generalized by Lee and Lifschitz [22] to disjunctive programs, can be adapted for our approach.

Acknowledgments

We thank Gerald Pfeifer for his help on experimental evaluation and fruitful discussions. We are also grateful to the reviewers of the paper as well as the reviewers of the preliminary conference versions for their comments and constructive suggestions for improvement.

A Integrated Program for Conformant Planning

The integrated program for the planning problem in Section 6.2, $Bomb_{plan} = Bomb_{guess} \cup Bomb_{check}{}'$, is given below. It has a single answer set $S = \{$ `dunk(0),-flush(0),flush(1),-dunk(1),...`$\}$ which corresponds to the single conformant plan P= `dunk`, `flush` as desired.

```
%%%% GUESS PART

% Timestamps:
  time(0). time(1).

% Guess a plan:
  dunk(T) v -dunk(T) :- time(T).
  flush(T) v -flush(T) :- time(T).
  :- flush(T), dunk(T).

%%%% REWRITTEN CHECK PART (after grounding)
%% 1. Create dynamically the facts for the program:

% armed(0) v -armed(0).
   lit(h,"armed(0)",1). atom("armed(0)","armed(0)").
   lit(h,"-armed(0)",1). atom("-armed(0)","armed(0)").

% armed(T1) :- armed(T), not -armed(T1), time(T), T1=T+1.
   lit(h,"armed(1)",2) :- time(0). atom("armed(1)","armed(1)").
   lit(p,"armed(0)",2) :- time(0).
   lit(n,"-armed(1)",2) :- time(0).

   lit(h,"armed(2)",3) :- time(1).  atom("armed(2)","armed(2)").
   lit(p,"armed(1)",3) :- time(1).
   lit(n,"-armed(2)",3) :- time(1).

% dunked(T1) :- dunked(T), T1=T+1.
   lit(h,"dunked(1)",4).  atom("dunked(1)","dunked(1)").
   lit(p,"dunked(0)",4).

   lit(h,"dunked(2)",5). atom("dunked(2)","dunked(2)").
   lit(p,"dunked(1)",5).

% dunked(T1) :- dunk(T), T1=T+1.
   lit(h,"dunked(1)",6) :- dunk(0).

   lit(h,"dunked(2)",7) :- dunk(1).

% armed(T1) v -armed(T1) :- dunk(T), armed(T), T1=T+1.
   lit(h,"armed(1)",8) :- dunk(0).
   lit(h,"-armed(1)",8) :- dunk(0). atom("-armed(1)","armed(1)").
```

```
   lit(p,"armed(0)",8) :- dunk(0).

   lit(h,"armed(2)",9) :- dunk(1).
   lit(h,"-armed(2)",9) :- dunk(1). atom("-armed(2)","armed(2)").
   lit(p,"armed(1)",9) :- dunk(1).

% -armed(T1) :- flush(1), dunked(T),T1=T+1.
   lit(h,"-armed(1)",10) :- flush(0). lit(p,"dunked(0)",10) :- flush(0).

   lit(h,"-armed(2)",11) :- flush(1). lit(p,"dunked(1)",11) :- flush(1).

% :- not armed(2).

%% 2. Optimized meta-interpreter

%% 2.1 -- program dependent part

   notok :- ninS("armed(0)"), ninS("-armed(0)").
   inS("armed(1)") :- inS("armed(0)"), ninS("-armed(1)"), time(0).
   inS("armed(2)") :- inS("armed(1)"), ninS("-armed(2)"), time(1).
   inS("dunked(1)") :- inS("dunked(0)").
   inS("dunked(2)") :- inS("dunked(1)").
   inS("dunked(1)") :- dunk(0).
   inS("dunked(2)") :- dunk(1).
   notok :- ninS("armed(1)"),ninS("-armed(1)"), inS("armed(0)"), dunk(0).
   notok :- ninS("armed(2)"),ninS("-armed(2)"),inS("armed(1)"), dunk(1).
   inS("-armed(1)") :- inS("dunked(0)"), flush(0).
   inS("-armed(2)") :- inS("dunked(1)"), flush(1).
   notok :- ninS("armed(2)").

%% 2.2 -- fixed rules

% Skipped, see QBF Encoding

%%% 3. constraint
   :- not notok.
```

B Integrated Program for Strategic Companies

The integrated program for the strategic companies problem instance in Section 6.3, $SC_{strategic}$ = $SC_{guess} \cup SC_{check}{}'$, is given below. It has two answer sets $S_1 = \{$`strat(barilla)`, `strat(saiwa)`, `strat(frutto)`, $\ldots\}$ and $S_2 = \{$`strat(barilla)`, `strat(panino)`, $\ldots\}$ which correspond to the strategic sets as identified above.

```
%%%% GUESS PART
  company(barilla). company(saiwa). company(frutto). company(panino).
  prod_by(pasta,barilla,saiwa).  prod_by(tomatoes,frutto,barilla).
  prod_by(wine,barilla,barilla). prod_by(bread,saiwa,panino).
  contr_by(frutto,barilla,saiwa,barilla).

%% Guess Program: Not necessarily minimal

  strat(X) v -strat(X) :- company(X).
```

```
   :-  prod_by(X,Y,Z), not strat(Y), not strat(Z).
   :-  contr_by(W,X,Y,Z), not strat(W),
       strat(X), strat(Y), strat(Z).

%%%% REWRITTEN CHECK PART (after grounding)
%% 1. Create dynamically the facts for the program:

% smaller :- -strat1(X).
  lit(h,"smaller",1). atom("smaller","smaller").
  lit(p,"-strat1(saiwa)",1).
  lit(h,"smaller",2). atom("smaller","smaller").
  lit(p,"-strat1(panino)",2).
  lit(h,"smaller",3). atom("smaller","smaller").
  lit(p,"-strat1(frutto)",3).
  lit(h,"smaller",4). atom("smaller","smaller").
  lit(p,"-strat1(barilla)",4).

% strat1(X) v -strat1(X) :- strat(X).
  lit(h,"strat1(saiwa)",5) :- strat(saiwa).    atom("strat1(saiwa)","strat1(saiwa)").
  lit(h,"-strat1(saiwa)",5) :- strat(saiwa).   atom("-strat1(saiwa)","strat1(saiwa)").
  lit(h,"strat1(panino)",6) :- strat(panino).  atom("strat1(panino)","strat1(panino)").
  lit(h,"-strat1(panino)",6) :- strat(panino). atom("-strat1(panino)","strat1(panino)").
lit(h,"strat1(frutto)",7) :- strat(frutto).    atom("strat1(frutto)","strat1(frutto)").
lit(h,"-strat1(frutto)",7) :- strat(frutto).   atom("-strat1(frutto)","strat1(frutto)").
lit(h,"strat1(barilla)",8) :- strat(barilla).  atom("strat1(barilla)","strat1(barilla)").
lit(h,"-strat1(barilla)",8) :- strat(barilla). atom("-strat1(barilla)","strat1(barilla)").

% For constraints, critical negative literals need to be represented (cf. OPT_mod)

% :- prod_by(X,Y,Z), not strat1(Y), not strat1(Z).
  lit(n,"strat1(saiwa)",10) :- prod_by(bread,saiwa,panino).
  lit(n,"strat1(panino)",10) :- prod_by(bread,saiwa,panino).
  lit(n,"strat1(frutto)",11) :- prod_by(tomatoes,frutto, barilla).
  lit(n,"strat1(barilla)",11) :- prod_by(tomatoes,frutto, barilla).
  lit(n,"strat1(barilla)",12) :- prod_by(wine,barilla,barilla).
  lit(n,"strat1(barilla)",13) :- prod_by(pasta,barilla,saiwa).
  lit(n,"strat1(saiwa)",13) :- prod_by(pasta,barilla,saiwa).

% :- contr_by(W,X,Y,Z), not strat1(W), strat1(X), strat1(Y), strat1(Z).
  lit(n,"strat1(frutto)",14) :- contr_by(frutto,barilla,saiwa,saiwa).

%% 2. Optimized meta-interpreter

%% 2.1 -- program dependent part
  inS("smaller") :- inS("-strat1(saiwa)").
  inS("smaller") :- inS("-strat1(panino)").
  inS("smaller") :- inS("-strat1(frutto)").
  inS("smaller") :- inS("-strat1(barilla)").
  notok :- ninS("strat1(saiwa)"),ninS("-strat1(saiwa)"),strat(saiwa).
  notok :- ninS("strat1(panino)"),ninS("-strat1(panino)"),strat(panino).
  notok :- ninS("strat1(frutto)"),ninS("-strat1(frutto)"),strat(frutto).
  notok :- ninS("strat1(barilla)"),ninS("-strat1(barilla)"),strat(barilla).
  notok :- ninS("smaller").
  notok :- ninS("strat1(saiwa)"),ninS("strat1(panino)").
  notok :- ninS("strat1(frutto)"),ninS("strat1(barilla)").
  notok :- ninS("strat1(barilla)").
```

```
  notok :- ninS("strat1(barilla)"),ninS("strat1(saiwa)").
  notok :- inS("strat1(barilla)"),inS("strat1(saiwa)"),ninS("strat1(frutto)").

%% 2.2 -- fixed rules

% Skipped, see QBF Encoding

%%% 3. constraint
   :- not notok.
```

C Ad Hoc Encoding for Quantified Boolean Formulas

The ad hoc encoding in [23] for evaluating a QBF of form $F = \exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n \, \Phi$, 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 3DNF, i.e. each $c_i = a_{i,1} \wedge \cdots \wedge a_{i,3}$ and $|a_{i,j}| \in \{x_1, \ldots, x_m, y_1 \ldots, y_n\}$, represents F by the following facts:

- `exists(`x_i`).` for each existential variable x_i;
- `forall(`y_j`).` for each universal variable y_j; and
- `term(`p_1,p_2,p_3,q_1,q_2,q_3`).` for each disjunct $c_j = l_{i,1} \wedge l_{i,2} \wedge l_{i,3}$ in Φ, where (i) if $l_{i,j}$ is a positive atom v_k, then $p_j = v_k$, otherwise p_j= `true`, and (ii) if $l_{i,j}$ is a negated atom $\neg v_k$, then $q_i = v_k$, otherwise q_i = `false`. For example, $term(x_1, \mathtt{true}, y_4, \mathtt{false}, y_2, \mathtt{false})$, encodes the term $x_1 \wedge \neg y_2 \wedge y_4$.

For instance, our sample instance from Section 6.1

$$\exists x_0 x_1 \forall y_0 y_1 (\neg x_0 \wedge \neg y_0) \vee (y_0 \wedge \neg x_0) \vee (y_1 \wedge x_0 \wedge \neg y_0) \vee (y_0 \wedge \neg x_1 \wedge \neg y_0)$$

would be encoded by the following facts:

```
exists(x0).  exists(x1).  forall(y1).  forall(y2).
term(true,true,true,x0,y0,false).
term(y0,true,true,x0,false,false).
term(y1,x0,true,y0,false,false).
term(y0,true,true,x1,y0,false).
```

These facts are conjoined with the following facts and rules:

```
t(true). f(false).
t(X) v f(X) :- exists(X).
t(Y) v f(Y) :- forall(Y).
          w :- term(X,Y,Z,Na,Nb,Nc),t(X),t(Y),t(Z),
               f(Na),f(Nb),f(Nc).
       t(Y) :- w, forall(Y).
       f(Y) :- w, forall(Y).
            :- not w.
```

The guessing part “initializes” the logical constants `true` and `false` and chooses a witnessing assignment σ to the variables in X, which leads to an answer set M_G for this part. The more tricky checking part then tests whether $\phi[X/\sigma(X)]$ is a tautology, using a saturation technique similar to our meta-interpreter.

D Ad Hoc Encodings for Strategic Companies

The first ad hoc encoding for Strategic Companies in [8], $adhoc_1$, solves the problem in a surprisingly elegant way by the following two rules conjoined to the facts representing the $prod\_by$ and $contr\_by$ relations:

```
strat(Y) v strat(Z) :- prod_by(X,Y,Z).
           strat(W) :- contr_by(W,X,Y,Z), strat(X), strat(Y), strat(Z).
```

Here, the minimality of answer sets plays together with the first rule generating candidate strategic sets and the second rule enforcing the constraint on the controls relation. It constitutes a sophisticated example of intermingled guess and check. However, this succinct encoding relies very much on the fixed number of producing and controlling companies; an extension to arbitrarily many producers and controllers seems not to be as easy as in our separate guess and check programs from Section 6.3.

The second ad hoc encoding from [8], $adhoc_2$, strictly separates the guess and checking parts, and uses the following rules and constraints:

```
strat(X) v -strat(X) :- company(X).
  :- prod_by(X,Y,Z), not strat(Y), not strat(Z).
  :- contr_by(W,X,Y,Z), not strat(W), strat(X), strat(Y), strat(Z).
  :- not min(X), strat(X).
  :- strat'(X,Y), -strat(Y).
  :- strat'(X,X).
min(X) v strat'(X,Y) v strat'(X,Z) :- prod_by(G,Y,Z),strat(X).
min(X) v strat'(X,C) :- contr_by(C,W,Y,Z), strat(X),
                         strat'(X,W), strat'(X,Y), strat'(X,Z).
         strat'(X,Y) :- min(X), strat(X), strat(Y), X!=Y.
```

Informally, the first rule and the first two constraints generate a candidate strategic set, whose minimality is checked by the remainder of the program. For a detailed explanation, we refer to [8].

References

[1] Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Journal of the Theory and Practice of Logic Programming*, 3:425–461, July/September 2003.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.

[3] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[4] Marco Cadoli, Thomas Eiter, and Georg Gottlob. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, May/June 1997.

[5] A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research 13*, 305–338, 2000.

[6] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[7] Jim Delgrande, Torsten Schaub, and Hans Tompits. plp: A Generic Compiler for Ordered Logic Programs. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, number 2173 in LNCS, pages 411–415. Springer, 2001.

[8] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[9] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Computing Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. *Journal of the Theory and Practice of Logic Programming*, 3:463–498, July/September 2003.

[10] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the $\text{DLV}^{\mathcal{K}}$ System. *Artificial Intelligence*, 144(1–2):157–211, March 2003.

[11] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic*, 5(2), April 2004.

[12] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. On Properties of Update Sequences Based on Causal Rejection. *Journal of the Theory and Practice of Logic Programming*, 2(6):721–777, 2002.

[13] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[14] Wolfgang Faber and Gerald Pfeifer. DLV homepage, since 1996. http://www.dlvsystem.com/.

[15] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[16] Michael Gelfond. Representing Knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic. Logic Programming and Beyond*, number 2408 in LNCS, pages 413–451. Springer, 2002.

[17] R. Goldman and M. Boddy. Expressive Planning and Explicit Knowledge. In *Proceedings AIPS-96*, pages 110–117. AAAI Press, 1996.

[18] G. Greco, S. Greco, and E. Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In *Proceedings 17th International Conference on Logic Programming (ICLP 2001)*, number 2237 in Lecture Notes in AI (LNAI), pages 348–364. Springer Verlag, 2001.

[19] Tomi Janhunen. Comparing the expressive powers of some syntactically restricted classes of logic porgrams. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*

- CL 2000, First International Conference, Proceedings, number 1861 in Lecture Notes in AI (LNAI), pages 852–866, London, UK, July 2000. Springer Verlag.

[20] Tomi Janhunen. On the effect of default negation on the expressiveness of disjunctive rules. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 93–106. Springer Verlag, September 2001.

[21] Tomi Janhunen, Ilkka Niemelä, Patrik Simons, and Jia-Huai You. Partiality and Disjunctions in Stable Model Semantics. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA*, pages 411–419. Morgan Kaufmann Publishers, Inc., 2000.

[22] Joohyung Lee and Vladimir Lifschitz. Loop Formulas for Disjunctive Logic Programs. In *Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03)*. Springer Verlag, December 2003.

[23] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning, 2004. To appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`.

[24] Nicola Leone, Riccardo Rosati, and Francesco Scarcello. Enhancing Answer Set Planning. In Alessandro Cimatti, Héctor Geffner, Enrico Giunchiglia, and Jussi Rintanen, editors, *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pages 33–42, August 2001.

[25] V. Lifschitz and H. Turner. Splitting a Logic Program. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[26] Vladimir Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138:39–54, 2002.

[27] Fangzhen Lin and Yuting Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, Canada, 2002. AAAI Press / MIT Press.

[28] Victor W. Marek and Jeffrey B. Remmel. On the expressibility of stable logic programming. *Journal of the Theory and Practice of Logic Programming*, 3:551–567, November 2003.

[29] Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[30] Drew McDermott. A Critique of Pure Reason. *Computational Intelligence*, 3:151–237, 1987. Cited in [5].

[31] Ilkka Niemelä. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[32] Christos H. Papadimitriou. A note on the expressive power of prolog. *Bulletin of the EATCS*, 26:21–23, 1985.

[33] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[34] Axel Polleres. *Advances in Answer Set Planning*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, Wien, Österreich, 2003.

[35] Alessandro Provetti and Son Tran Cao, editors. *Proceedings AAAI 2001 Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, Stanford, CA, March 2001. AAAI Press.

[36] T. Przymusinski. On the Declarative and Procedural Semantics of Logic Programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.

[37] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[38] Chiaki Sakama and Katsumi Inoue. An abductive framework for computing knowledge base updates. *Journal of the Theory and Practice of Logic Programming*, 3:671–713, November 2003.

[39] J.S. Schlipf. The Expressive Powers of Logic Programming Semantics. *Journal of Computer and System Sciences*, 51(1):64–86, 1995. Abstract in Proc. PODS 90, pp. 196–204.

[40] Hudson Turner. Polynomial-Length Planning Spans the Polynomial Hierarchy. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA)*, number 2424 in Lecture Notes in Computer Science, pages 111–124, Cosenza, Italy, September 2002. Springer.