Artificial Intelligence 290 (2021) 103402

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Pruning external minimality checking for answer set programs using semantic dependencies ☆

Thomas Eiter, Tobias Kaminski *

Institute of Logic and Computation, Knowledge-Based Systems Group, Technische Universität Wien, Vienna, Austria

ARTICLE INFO

Article history:
Received 26 January 2020
Received in revised form 3 September 2020
Accepted 11 October 2020
Available online 16 October 2020

Keywords:
Knowledge representation and reasoning
Logic programming
Answer set programming with external source access
Semantic dependencies

ABSTRACT

Answer set programming (ASP) has become an increasingly popular approach for declarative problem solving. In order to address the needs of applications, ASP has been extended in different approaches with means for interfacing the outside world, of which HEX programs are one of the most powerful such extension that provides API-style interfaces to access arbitrary external sources of information and computation, respectively. Adhering to the principle of founded derivation, computing answer sets of HEX programs requires an external (e-) minimality check for answer set candidates in order to prevent cyclic justifications via external sources. Due to the generic nature of external sources, the check can be a bottleneck in practice. To mitigate this, various optimizations have been developed previously, including the use of syntactic information about atom dependencies in order to detect cases when an e-minimality check can be avoided. However, the approach largely over-approximates the real dependencies due to the black-box nature of external sources. We thus consider in this work the use of semantic information for achieving better approximations. To this end, we introduce input-output (io-) dependencies for external sources, which intuitively link the occurrence of values in the result of a call to an external source to the occurrence of values in the input provided to this call. It appears that disposing of information about io-dependencies significantly increases the potential for pruning e-minimality checks, and an empirical evaluation exhibits a clear benefit of this approach. Moreover, we study semantic and computational properties of io-dependencies and provide algorithms for constructing and optimizing sets of io-dependencies. Our work aims at laying some foundations for the use of semantic dependency information in external source access from ASP. The results are not limited to HEX programs, but may analogously be deployed to other approaches that integrate external sources into ASP, such as CLINGO or WASP with external propagators. Furthermore, the results may be applied in other parts of the HEX program evaluation pipeline as well.

Contents

☆ This is a Fast Track Invited Paper. This article is a revised and extended version of the paper presented at the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019), June 3-7, 2019, Philadelphia, PA, USA.

* Corresponding author.
E-mail addresses: eiter@kr.tuwien.ac.at (T. Eiter), kaminski@kr.tuwien.ac.at (T. Kaminski).

https://doi.org/10.1016/j.artint.2020.103402

1. Introduction

Declarative problem solving has been gaining momentum in the last decade, with approaches such as *SAT solving*, *constraint programming* (*CP*), and *answer set programming* (*ASP*) as well-known representatives. The latter, which is based on the *stable model semantics* of logic programs [43,44], has become popular through the availability of efficient solvers and the expressiveness that is offered by a rich variety of ASP language constructs and extensions, which have been exploited in a great manifold of applications, cf. [8,30,34]. The interest in ASP is further witnessed by recent books [38,42] and special issues of journals [9,71,6].

ASP as such has not been conceived as a general purpose programming language and is more geared towards solving particular problems within the context of comprising applications; on the other hand, ASP programs often require means to interact with the outside world, in order to access or release information through well-defined interfaces that may link them also to other programs and systems.

Several works have considered the use of ASP inside programming languages and systems, among them the *DLV Java Wrapper* [67], JDLV [35], the scripting facilities featured by CLINGO 4 [39], ASAP [73] and the *embASP* framework [36,14], which aims at the integration of logic formalisms in external systems for generic applications; see [14] for more discussion.

As regards the access to the outside world from ASP programs, many dedicated extensions of ASP towards the integration of specific external sources have been developed. Examples are extensions of the DLV solver with interfaces to relational databases [77] and ontologies [68], constraint ASP for integrating ASP with CP as realized e.g. in CLINGCON [63], which combines CLASP, GRINGO, and the constraint solver GECODE, or ASP extended with SMT as e.g. in EZSMT [75], DINGO [51], and ASPMT [55]; see [59] for an overview of systems combining ASP with CP and other theories.

On the other hand, also ASP extensions with generic interfaces have been proposed, such as DLV-EX [11] to access user-defined predicates while allowing for *value invention*, i.e., that new terms can be introduced into the logic program. The CLINGO system [39] similarly allows to import user-defined predicates, implemented in the scripting languages *Lua* or *Python*, as customizable built-ins. Furthermore, the CLINGO 5 release [37] takes this further by providing generic interfaces for integrating theory solving into ASP. Besides CLINGO, the WASP solver was extended with support for general external *Python* propagators [19,18].

HEX programs. One of the most powerful extensions of ASP of the second kind is the HEX *formalism* [27], which provides *external atoms* that can interface arbitrary external sources in an API-style fashion at a high level of abstraction. This can be flexibly used in different ways, from user-defined built-ins as in CLINGO over accessing information in different formats

(e.g., thesauri, databases, RDF repositories), to invoking computations of any kind as a service (e.g., reasoning on knowledge bases, path computations, planners or constraint solving). For example, an external atom $\&concat[X, Y](Z)$ can be used to concatenate strings, where the elements inside the square brackets can be viewed as inputs to an external source, which is named 'concat' and able to concatenate strings, and the element inside the parentheses is an output value computed by the external source based on the respective inputs. To distinguish external predicates from regular predicates, they are prefixed by an ampersand. The external atom could be used in a rule

$$fullname(X) \leftarrow \&concat[X, Y](Z), firstname(X), lastname(Y) \tag{1}$$

as customized built-in for strings. Notably, external atoms may also have predicate input. This and the fact that a bidirectional information exchange between the rules and the external source is possible makes the formalism more general than DLV-EX and CLINGO's external predicates. For example, in a rule

$$closeCity(X) \leftarrow \&closeTo[city](X), location(X) \tag{2}$$

the external atom $\&closeTo[city](X)$ may output all cities that are located close to cities in the predicate *city*, whose extension in the interpretation considered will be given to the external source for evaluation. This extension may be given by facts but can also be defined by other rules.

Supported by sophisticated solvers [25,66,72], the versatility of HEX programs has been exploited in many areas ranging from *semantic web* and query answering applications [49,65,31] over game playing [13] to planning in robotics [30,32]; we refer to [27,29] for a more extensive account of applications. Particularly powerful is the use of external sources in a recursive fashion, like web crawling, exploring routes along a map, etc. However, such use requires careful consideration, since mutually supporting derivations are possible. For example, if the locations for the rule (2) are *osaka*, *kobe*, *bratislava* and *vienna*, and the rule

$$city(X) \leftarrow closeCity(X) \tag{3}$$

as well as the fact $city(osaka)$ are added, then only the atom $city(kobe)$ should be contained in an answer set in addition. Even though Bratislava and Vienna are located close to each other, the atoms $city(bratislava)$ and $city(vienna)$ can only cyclically support each other via the two rules (2) and (3), and the external atom in (2).

For *Horn-style* programs with no disjunction and no negation, such unfounded mutual support can sometimes be avoided and the unique answer set can be computed bottom-up by fixpoint iteration, as customary for Horn logic programs. For the general class of HEX programs, algorithms have been developed that resort to using an ordinary ASP solver as follows: a HEX program Π at hand is rewritten to an ordinary ASP program $\hat{\Pi}$, where external atoms are replaced by auxiliary atoms whose valuations are guessed. Then answer sets of this *replacement program* $\hat{\Pi}$ are computed and each is checked for correctness of the external evaluation guess, yielding so called *compatible sets*. In the last step, the compatible sets are checked for foundedness (external minimality) with respect to the original program Π.

Thus, an important difference to ordinary ASP evaluation is that an *external* (*e-*) *minimality check* is needed for computing answer sets of HEX programs to avoid unfounded support by external atoms; in the example above, this check would eliminate spurious answer sets containing the mutually supporting atoms $city(bratislava)$ and $city(vienna)$. We note that in the CLINGO extensions, an external minimality check is disregarded; thus expressing recursion through external predicates is not well-supported. Consequently, e.g. recursive aggregates as in [33], which are easily definable in HEX with external atoms, cannot be readily expressed via external predicates in CLINGO.

Unfortunately, performing the e-minimality check efficiently is highly non-trivial as it is co-NP-complete already for ground Horn programs with polynomial external atoms [24]. Thus, special techniques have been developed to either make the test (which is based on *unfounded sets*) more efficient, or to avoid the test altogether. For example, if the rule (3) is not added above, cyclic support via the external atom can be ruled out independent from the external semantics.

Taking dependencies among atoms into account is a crucial technique for the evaluation of logic programs, which is also important for modularity [52]; they are represented in a dependency graph, which can be processed efficiently using graph algorithms. Cyclic dependencies due to mutual recursion such as by rules $a \leftarrow b$ and $b \leftarrow a$, are problematic; in absence of such cycles, derivations are founded.

Based on this observation, a syntactic criterion was presented in [24] for deciding whether the e-minimality check can be skipped for a program, by checking whether no cyclic dependencies through external atoms, called *external* (*e-*) *cycles*, occur in the program. However, this test is rather coarse and due to the black-box nature of external atoms, there are cases in which potential cyclic dependencies are accounted for which do not exist. The reason is that in lack of further knowledge, one has to assume that every atom in the input of an external atom may be relevant for computing a particular output; this leads to a large number of edges in the dependency graph. For instance, in the example above, the dependency graph would contain an edge representing that the output *bratislava* of the external atom $\&closeTo[city](X)$ depends on $city(osaka)$. Skipping e-minimality checks in more cases is of special interest as it can often result in significant speedups.

Semantic dependencies. The situation may drastically improve if one has some knowledge about how input relates to output, and in particular on which part of the input a particular output depends. If in the example above, it is known that

only *vienna* is close to *bratislava*, then the aforementioned dependency edge can be dropped. Furthermore, if a different external atom &*closeWest*[*city*](*X*) is used in the example (only retrieving cities located close *to the west* of input cities), cyclic support can be excluded based on the external source semantics. However, this cannot be detected by a syntactic criterion, and the e-minimality check must be performed by the traditional HEX evaluation approach. Moreover, applying a semantic criterion is challenging because, previously, external atoms have largely been considered as black-boxes that conceal semantic dependencies.

For this reason, we develop a new approach for pruning e-minimality checking that also exploits semantic dependencies. It relies on additional information about *input-output* (*io-*) *dependencies* of external atoms, which may be provided by a user, or even generated automatically. While different possibilities for expressing dependencies can be imagined, we take here a data perspective in which value occurrences matter, i.e., under which conditions does a particular value occur in the output, and in a more fine-grained view, at which particular position. This is conditioned on the occurrence of values in the input: only atoms in the input where certain values occur at certain positions are relevant for determining the output value.

In the example above, one may state that whether *kobe* is in the output of the external atom &*closeWest*[*city*](*X*) only depends on whether *city*(*osaka*) is true or not. In terms of value dependency, this means that the occurrence of *kobe* in the output depends on the occurrence of *osaka* in the input, which, in a formal language that is introduced in this paper, is more precisely stated by an expression $\langle 1, 1:\{osaka\}, 1:kobe\rangle$, where "1" refers to the first input predicate (in this case, the only one), "$1:\{osaka\}$" to the value at the first argument of that input predicate, and "$1:kobe$" to the value at the first (again, the only) argument of the output; in this particular case, the position information is redundant, and one could simply write $\langle\{osaka\}, kobe\rangle$.

Hidden io-dependencies seem to be common in applications involving recursive processing, e.g. over external graphs or semantic web data, where output values can be traced back to certain input values, e.g. in graph-like structures from a node to ancestors of the node. Furthermore, in other cases values in the input may be passed through and occur in the output as well; often the user even might be aware of this. For example, an external atom &*topLiveable*[*city*](*X*) may single out from the cities in the input the one which is most liveable according to some ranking.

We note that io-dependencies have been considered in other contexts (see the discussion of related work in Section 6.1) and are important for reducing costs and increasing efficiency, as well as to obtain a clearer picture of semantic properties.

Contributions. The main contributions of our work can be briefly summarized as follows.

(1) We provide a novel formalization of io-dependencies that encode semantic dependency information, and we show under which condition they can safely be used for pruning the e-minimality check. Formally, we represent io-dependencies by expressions $\langle i, j : J, k : c\rangle$, which state that for the occurrence of c at position k of the output of an external atom, only those facts in the i-th input predicate with a value from J at position j matter; cf. the io-dependency $\langle 1, 1:\{osaka\}, 1:kobe\rangle$ from above. Considering such data dependencies is particularly effective as they are at the ground level and allow to catch individual relationships; as an attractive feature, dependency information may be incomplete and added flexibly. Abstract, generic dependencies can be provided on top with a grounding semantics as usual. The format is a compromise between efficient realizability and semantic accuracy: tracking all tuples or combinations of tuples on which an output depends might be too expensive (Section 6.2 provides more discussion). We consider io-dependencies in different contexts: with no information about a program in use (called *universal io-dependencies*) versus with information about the possible input of an external atom (*relativized io-dependencies*), given by an *envelope* (a superset of the atoms that may occur in an answer set).

(2) We study semantic properties of io-dependencies, where we focus on a property called *faithfulness*. Roughly speaking, a set of io-dependencies has the faithfulness property, if it captures all true dependencies between atoms. We also study operations to change sets of io-dependencies that preserve faithfulness, such as removing items from a set of io-dependencies and intersecting sets of io-dependencies. These operations are then used in algorithms to optimize sets of io-dependencies, in order to remove redundancies and make them *tight*, which means that no further dependency information can be removed without violating the faithfulness property. Moreover, we show that io-dependencies enjoy an intersection property, i.e. the intersection of two faithful sets of io-dependencies is also faithful. As a consequence, semantically, a unique tightest set of io-dependencies exists, which results in the best approximation of the real atom dependencies and allows for the most pruning of e-minimality checks during the evaluation of HEX programs. Similar results are derived for relativized io-dependencies.

(3) We characterize the computational complexity of a key problem in io-dependency management, viz. to decide whether a given set of io-dependencies for an external atom is faithful. As it turns out, this problem is for universal io-dependencies co-NEXPTIME-time complete in general and co-NP-complete if the input predicates have arities bounded by a constant; under certain restrictions, tractability of faithfulness checking is obtained. Notably, faithful io-dependencies are a sufficient but not a necessary criterion for being able to skip the e-minimality check. Testing the latter property is shown to be co-NEXPTIME$^{\text{NP}}$-complete (resp. Π_3^p-complete), and thus much more expensive in the worst case.

For relativized io-dependencies, the complexity of faithfulness checking depends on the particular envelope used, and on whether the latter is given in the input or not. For the envelope that consists of the brave consequences of the program Π, the test is co-NEXPTIME$^{\text{NP}}$-complete (resp. Π_3^p-complete), and thus as expensive as the semantic skipping test. On the other hand, for the *Horn envelope*, which is obtained from a transformation of a HEX programs Π to an ordinary Horn logic program, the complexity drops and the test is co-NEXPTIME-complete (resp. co-NP-complete).

In the course of our analysis, we discuss how to reduce general HEX programs to Horn-style HEX programs in polynomial time, by emulating negation and disjunction with external atoms. This complements the result of Faber et al. [33] that for Horn-style propositional ASP programs with recursive aggregates, deciding whether an answer set exists is Σ_2^p-complete, and is interesting in its own right.

(4) We describe an implementation of the new technique that has been integrated into the DLVHEX solver, and we report on an experimental evaluation of the approach on a suite of benchmark problems. In the experiments, the interaction of io-dependency pruning with previously established optimization techniques is studied, among them external evaluation with candidate models, with partial assignments, and with partial assignments during minimality checking. While the results show that there is no clear winner among the configurations, they confirm the advantage of exploiting io-dependencies which can yield significant performance gains without incurring much overhead otherwise.

The results of this paper contribute to the challenging goal of increasing the efficiency of ASP programs with external atoms with new techniques. They are not only applicable to HEX programs, but may also be employed analogously for other approaches that integrate external sources into ASP, such as CLINGO [37], if external cyclic support is not desired, respectively for arguing that answer sets obtained are founded and free of cyclic derivations through external predicates; similarly for external propagators [19,18]. Furthermore, the results may be applied in other tasks of the HEX program evaluation pipeline as well.

Organization. The remainder of this article is organized as follows. The next section recalls notions from ASP and HEX programs as needed for this work, and reviews the approach for evaluating HEX programs based on a program rewriting. After that, we introduce, in Section 3, io-dependencies for pruning external minimality checks, and present semantic properties and optimization algorithms. Furthermore, we extend the results to relativized io-dependencies. Section 4 is devoted to analyzing the intrinsic complexity of faithfulness checking, which is the key problem in io-dependency management; we consider various settings, for universal and relativized io-dependencies using different natural envelopes. In Section 5, implementation and experimental evaluation of the approach are considered, which is followed by a discussion in Section 6 including related work, generalizations and other applications of the approach. The final Section 7 gives a short summary and outlines issues for future research.

2. Preliminaries

We assume disjoint sets $\mathcal{P}$, $\mathcal{C}$, $\mathcal{X}$ and $\mathcal{V}$ of predicates, constants, external predicates (prefixed with '&') and variables, respectively. Each $p \in \mathcal{P}$ has fixed positive arity $ar(p) > 0$, and each $\&g \in \mathcal{X}$ has fixed input arity $ar_I(\&g) \geq 0$ and output arity $ar_O(\&g) > 1$, respectively. An atom is of the form $p(\vec{t})$, where $p \in \mathcal{P}$ and $\vec{t} = t_1, \ldots, t_\ell$ is a list of *terms* $t_i \in \mathcal{C} \cup \mathcal{V}$; it is *ground*, if no variable occurs in it.

A (signed) literal is a positive or a negative literal $\mathbf{T}p(\vec{c})$ or $\mathbf{F}p(\vec{c})$, where $p(\vec{c})$ is a ground atom. An *assignment* $\mathbf{A}$ over a set $\mathcal{A}$ of ground atoms is a set of literals such that for each $a \in \mathcal{A}$, either $\mathbf{T}a \in \mathbf{A}$ or $\mathbf{F}a \in \mathbf{A}$, where $\mathbf{A}(a) = \mathbf{T}$ (i.e., $\mathbf{A}$ satisfies a, denoted $\mathbf{A} \models a$) if $\mathbf{T}a \in \mathbf{A}$, and $\mathbf{A}(a) = \mathbf{F}$ (i.e., $\mathbf{A}$ falsifies a, denoted $\mathbf{A} \not\models a$) otherwise.

To keep the treatment simple, we disregard in this article 0-ary predicates p (i.e., propositional letters) and 0-ary Boolean external predicates $\&g[\vec{p}]$ (i.e., $ar_O(\&g) = 0$); they can be emulated by using an atom $p'(c_0)$ for p respectively an external atom $\&g'[\vec{p}](c_0)$ for $\&g[\vec{p}]()$, where p' and $\&g'$ are fresh predicates and c_0 is some arbitrary constant.

2.1. HEX programs

HEX *programs* extend answer set programs with *external atoms* in rule bodies (cf. [27] for more details).

Syntax. An *external atom* is of the form $\&g[\vec{X}](\vec{Y})$, where $\&g \in \mathcal{X}$, $\vec{X} = X_1, \ldots, X_k$, with $k = ar_I(\&g)$, are input parameters, which are terms and/or predicates, and $\vec{Y} = Y_1, \ldots, Y_l$, with $l = ar_O(\&g)$, are output terms. An external atom is *ground* if $\vec{X} = X_1, \ldots, X_k$ are predicates and constants and $\vec{Y} = Y_1, \ldots, Y_l$ are constants. Given a ground external atom $\&g[\vec{X}](\vec{Y})$, we call $\&g[\vec{X}]$ a *ground external* (*ge-*)*predicate*. For uniformity and without loss of generality, constants c in the input list $\vec{p}$ of a ground external atom $\&g[\vec{p}](\vec{c})$ are viewed as unary predicates p_c for which the single fact $p_c(c)$ exists.

Example 1. The external atom $\&closeTo[city](X)$ mentioned in the introduction has one input argument, which intuitively is a set of cities, given by the predicate *city*. The single output argument stands for objects X that are intuitively close to some city in *city* – this intuitive view of course has to be made formally precise in the semantics. The ground instance $\&closeTo[city](vienna)$ then intuitively describes that *vienna* is close to some city in *city*.

For another example, the identity atom $\&id[p](X)$ may output all constants X that are contained in p; that is, $\&id[p](c)$ is true iff $p(c)$ is true.

Definition 1 *(HEX Program).* A HEX program Π is a set of rules of the form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n, \tag{4}$$

where each a_i, $1 \leq i \leq k$, is an atom and each b_j, $1 \leq j \leq n$, is either an ordinary atom or an external atom.

Given a rule r, $H(r) = \{a_1, \ldots, a_k\}$ is its *head*, $B(r) = \{b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\}$ its *body*, and $B^+(r) = \{b_1, \ldots, b_m\}$ resp. $B^-(r) = \{b_{m+1}, \ldots, b_n\}$. A rule r is *ground*, if every atom in r is ground; a program Π is ground, if every rule r in Π is ground. A ground rule r is a *fact* if $k = 1$ and $n = 0$. As usual, *constraints* can be regarded as rules of the form $f \leftarrow B, \text{not } f$, written as $\leftarrow B$, where f is a distinct atom occurring only in such rules; informally, $\leftarrow B$ eliminates all models in which B is satisfied.

Semantics. As for ordinary ASP programs, the semantics of HEX programs is defined via grounding, where each rule r in a program Π is replaced by its grounding $grnd_{\mathcal{C}}(r)$, which is the set of all ground instances r' of r over $\mathcal{C}$, i.e., all rules r' obtained by substituting for each variable X in r some constant from $\mathcal{C}$; the grounding of Π is thus $grnd_{\mathcal{C}}(\Pi) = \bigcup_{r \in \Pi} grnd_{\mathcal{C}}(r)$. The subscript $\mathcal{C}$ may be omitted if $\mathcal{C}$ is clear from the context.

Following Eiter et al. [28], let $\mathbb{A}_{\mathcal{P},\mathcal{C}}$ denote the set of all assignments $\mathbf{A}$ over the *Herbrand base* induced by $\mathcal{P}$ and $\mathcal{C}$, which is the set of all ground atoms $p(\vec{c})$ with predicate $\mathcal{P}$ and constants from $\mathcal{C}$.

The semantics of a ground external atom $\&g[\vec{p}](\vec{c})$, wrt. an assignment $\mathbf{A}$ is given by the value of a $1{+}k{+}l$-ary decidable *two-valued (Boolean) oracle function*

$$f_{\&g} \colon \mathbb{A} \times (\mathcal{P} \cup \mathcal{C})^k \times \mathcal{C}^l \to \{\mathbf{T}, \mathbf{F}\}$$

where $k = ar_I(\&g)$ and $l = ar_O(\&g)$ are the lengths of $\vec{p}$ and $\vec{c}$, respectively. Thus, $\&g[\vec{p}](\vec{c})$ is true relative to $\mathbf{A}$ (informally, $\vec{c}$ is an output of $\&g$ for input $\vec{p}$, denoted $\mathbf{A} \models \&g[\vec{p}](\vec{c})$), if $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = \mathbf{T}$ and false ($\vec{c}$ is not an output of $\&g$ for input $\vec{p}$, denoted $\mathbf{A} \not\models \&g[\vec{p}](\vec{c})$), otherwise.

Example 2 *(Example 1, cont'd).* Suppose that $\mathcal{C} = \{osaka, kobe, bratislava, vienna\}$, and that for any assignment $\mathbf{A}$ we have that

$$\mathbf{A} \models \&closeTo[city](c) \text{ iff } \begin{cases} c = bratislava \wedge \mathbf{T}city(vienna) \in \mathbf{A}, \\ c = vienna \wedge \mathbf{T}city(bratislava) \in \mathbf{A}, \\ c = kobe \wedge \mathbf{T}city(osaka) \in \mathbf{A}, \\ c = osaka \wedge \mathbf{T}city(kobe) \in \mathbf{A}; \end{cases}$$

that is, $f_{\&closeTo[city]}$ formalizes the proximity relation. Similarly,

$$\mathbf{A} \models \&closeWest[city](c) \text{ iff } \begin{cases} c = vienna \wedge \mathbf{T}city(bratislava) \in \mathbf{A}, \\ c = kobe \wedge \mathbf{T}city(osaka) \in \mathbf{A}. \end{cases}$$

Furthermore, for any $c \in \mathcal{C}$,

$$\mathbf{A} \models \&id[p](c) \text{ iff } \mathbf{T}p(c) \in \mathbf{A}.$$

Satisfaction of ASP rules and programs [44] is extended to HEX rules and programs in the expected way. That is, an assignment $\mathbf{A}$ satisfies a ground rule r, if either (i) $\mathbf{A} \models a$ for some $a \in H(r)$, or (ii) $\mathbf{A} \not\models b$ for some $b \in B^+(r)$, or (iii) $\mathbf{A} \models b$ for some $b \in B^-(r)$. Furthermore, $\mathbf{A}$ satisfies an arbitrary rule r, if it satisfies each rule r' in $grnd(r)$, and an arbitrary program Π, if $\mathbf{A}$ satisfies each rule $r \in \Pi$.

The answer sets of a ground HEX program Π are now defined as follows. Let the *FLP-reduct* [33] of Π with respect to an assignment $\mathbf{A}$ be the set $f\Pi^{\mathbf{A}} = \{r \in \Pi \mid \mathbf{A} \models b, \text{ for all } b \in B(r)\}$ of all rules whose body is satisfied by $\mathbf{A}$. Furthermore, let for assignments $\mathbf{A}_1, \mathbf{A}_2$ denote $\mathbf{A}_1 \leq \mathbf{A}_2$ that $\{\mathbf{T}a \in \mathbf{A}_1\} \subseteq \{\mathbf{T}a \in \mathbf{A}_2\}$.

Definition 2 *(Answer Set).* An assignment $\mathbf{A}$ is an answer set of a ground HEX program Π, if $\mathbf{A}$ is a $\leq$-minimal model of $f\Pi^{\mathbf{A}}$, and an answer of an arbitrary HEX program Π, if it is an answer set of $grnd(\Pi)$.

Example 3 *(Example 2 cont'd).* Consider the program Π with the following rules and facts:

$$city(X) \leftarrow closeCity(X).$$
$$closeCity(X) \leftarrow \&closeTo[city](X), location(X).$$
$$city(vienna).$$
$$location(bratislava).$$

Then, the assignment $\mathbf{A}_0$ in which in addition to $city(vienna)$ and $location(bratislava)$ also $closeCity(bratislava)$ and $city(bratislava)$ are true is an answer set of Π. Note that the FLP-reduct of $grnd(\Pi)$ wrt. $\mathbf{A}_0$ contains besides the facts the ground rules

$$city(bratislava) \leftarrow closeCity(bratislava).$$
$$closeCity(bratislava) \leftarrow \&closeTo[city](bratislava), location(bratislava).$$

It is easily checked that $\mathbf{A}_0$ is a $\leq$-minimal model of $f grnd(\Pi)^{\mathbf{A}_0}$.

For $\mathcal{C} = \{a\}$, the program $\Pi = \{p(X) \leftarrow \&id[p](X)\}$, has the answer set $\mathbf{A}_1 = \emptyset$; indeed, $grnd(\Pi) = \{p(a) \leftarrow \&id[p](a)\}$, and $\mathbf{A}_1 = \emptyset$ a $\leq$-minimal model of $fgrnd(\Pi)^{\mathbf{A}_1} = \emptyset$.

Answer sets of HEX programs have similar properties as answer sets of ASP programs. For example, every answer set of a HEX program Π is a minimal model of Π, and every atom a that is true in some answer set $\mathbf{A}$ of Π must be supported, i.e., some ground instance r' of a rule r in Π must exist such that $\mathbf{A} \models B(r)$ and $\mathbf{A} \not\models H(r) \setminus \{a\}$. We refer to [27] for a more extensive discussion, but highlight here a specific property that is important with respect to external atom evaluation.

Let $A(\Pi)$ denote the set of all (ordinary) ground atoms that occur in a ground HEX program Π.

Proposition 1. *For every answer set* $\mathbf{A}$ *of a ground* HEX *program* Π, *it holds that* $\{a \mid Ta \in \mathbf{A}\} \subseteq A(\Pi)$.

Consequently, for the evaluation of a ground HEX program (including the input of external atoms $\&g[\vec{p}](\vec{c})$), we can confine to the atoms occurring in Π and tacitly assume that all other atoms are false. In fact, due to supportedness we could further narrow down to the atoms occurring in rule heads of Π.

2.2. HEX *program evaluation*

As usual, we make the assumption that the value of $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c})$ depends only on the restriction of $\mathbf{A}$ to the predicates in $\vec{p}$; that is, only the valuation of the input predicates matters for determining the output values of an external atom.

Next we assume that programs Π have equivalent finite groundings, i.e., some finite program $\Pi' \subseteq grnd(\Pi)$ that has the same answer sets as Π can be effectively computed from Π. This can be ensured by suitable safety notions for rules and programs, which have been discussed in [23]. Notice that this issue is nontrivial and not assured in general, as external atoms may introduce new values from $\mathcal{C}$ that do not occur in the program Π (known as *value invention*), which may enforce that only infinite equivalent groundings do exist if $\mathcal{C}$ is infinite (e.g., if it consists of all finite strings, or all natural numbers). Under this proviso, technically the sets $\mathcal{P}$ and $\mathcal{C}$ can be restricted to the finite subsets $\mathcal{P}' \subseteq \mathcal{P}$ of predicates and $\mathcal{C}' \subseteq \mathcal{C}$ of constants, respectively, that occur in Π'; we thus assume that $\mathcal{P}$ and $\mathcal{C}$ are finite.

2.2.1. *Guessing program*

A ground HEX program Π can be transformed to an ordinary ground program $\hat{\Pi}$ called the *guessing program*, as follows:

- replace each external atom $\&g[\vec{p}](\vec{c})$ in Π by an ordinary *replacement atom* $e_{\&g[\vec{p}](\vec{c})}$; and
- add a rule $e_{\&g[\vec{p}](\vec{c})} \vee ne_{\&g[\vec{p}](\vec{c})} \leftarrow$ that guesses its evaluation.

A similar guessing transformation can be done for non-ground programs Π, where variables are removed from the input lists (by grounding); to obtain safe rules, domain predicates may be used.

An ordinary ASP solver can then be employed to compute the answer sets of $\hat{\Pi}$; each such answer set $\hat{\mathbf{A}}$ is a *candidate model*. If moreover all truth values for atoms $e_{\&g[\vec{p}](\vec{c})}$ correspond to $f_{\&g}(\hat{\mathbf{A}}, \vec{p}, \vec{c})$, then $\hat{\mathbf{A}}$ is called a *compatible set*. Notably, each answer set $\mathbf{A}$ of Π has a corresponding compatible set.

Example 4 *(Example 3 cont'd).* For the naive grounding of the cities-program Π over $\mathcal{C}$, the guessing program $\hat{\Pi}$ consists of the rules

$$\begin{aligned}
city(c) &\leftarrow closeCity(c) && c \in \mathcal{C}.\\
closeCity(c) &\leftarrow e_{\&closeTo[city]}(c), location(c) && c \in \mathcal{C}.\\
city(vienna).&\\
location(bratislava).&\\
e_{\&closeTo[city]}(c) \vee ne_{\&closeTo[city]}(c) &\leftarrow && c \in \mathcal{C}.
\end{aligned}$$

Then, the extension $\hat{\mathbf{A}}_0$ of the assignment $\mathbf{A}_0$ which contains $\mathbf{T}e_{\&closeTo[city]}(c)$ for $c = vienna, bratislava$ and $\mathbf{T}ne_{\&closeTo[city]}(c)$ for $c = kobe, osaka$ is a compatible set; it is in fact the only compatible set, and gives rise to the (unique) answer set of Π.

Still, the projection of a compatible set $\hat{\mathbf{A}}$ to the original signature is not always an answer set due to the possibility of cyclic support via external atoms.

Example 5 *(Example 3 cont'd).* For the program Π with the identity atom, the guessing program is

$$\hat{\Pi} = \{\, p(a) \leftarrow e_{\&id[p]}(a);\;\; e_{\&id[p]}(a) \vee ne_{\&id[p]}(a) \leftarrow \};$$

it has the two answer sets $\hat{\mathbf{A}}_1 = \emptyset$ and $\hat{\mathbf{A}}_2 = \{\mathbf{T}p(a), \mathbf{T}e_{\&id[p]}(a)\}$. Here, the projection $\mathbf{A}_1$ is a $\leq$-minimal model of $f\Pi^{\mathbf{A}_1} = \emptyset$, but the projection $\mathbf{A}_2$ is not a $\leq$-minimal model of $f\Pi^{\mathbf{A}_2} = \Pi$ since $\emptyset \leq \mathbf{A}_2$ is a smaller model.

Consequently, an e-minimality check wrt. $f\Pi^{\mathbf{A}}$ is needed for finding answer sets of HEX programs.

2.2.2. External minimality check

A direct way to ensure e-minimality of the projection $\mathbf{A}$ of a compatible set $\hat{\mathbf{A}}$ for a HEX program Π wrt. $f\Pi^{\mathbf{A}}$ consists in explicitly constructing $f\Pi^{\mathbf{A}}$ and checking that it has no model $\mathbf{A}'$ such that $\mathbf{A}' \leq \mathbf{A}$. However, an alternative check based on *unfounded sets* can be significantly faster as it avoids the explicit generation of models of $f\Pi^{\mathbf{A}}$ [24]. An unfounded set of a HEX program Π wrt. an assignment $\mathbf{A}$ is a set of atoms that can be jointly set to false without violating any rule in Π as they only mutually support each other wrt. $\mathbf{A}$. Formally, for an assignment $\mathbf{A}$ and a set of atoms U, we denote by $\mathbf{A}\dot{\cup}\neg.U = (\mathbf{A} \setminus \{\mathbf{T}a \mid a \in U\}) \cup \{\mathbf{F}a \mid a \in U\}$ the assignment obtained by setting all atoms from U in $\mathbf{A}$ to false.

Definition 3 *(Unfounded Set).* For a ground HEX program Π and an assignment $\mathbf{A}$, $U \subseteq A(\Pi)$ is an *unfounded set* for Π wrt. $\mathbf{A}$ if, for each rule $r \in \Pi$ with $H(r) \cap U \neq \emptyset$, at least one of the following holds:

(i) some literal of $B(r)$ is false wrt. $\mathbf{A}$,
(ii) some literal of $B(r)$ is false wrt. $\mathbf{A}\dot{\cup}\neg.U$, or
(iii) some atom of $H(r) \setminus U$ is true wrt. $\mathbf{A}$.

The following result was established in [24].

Proposition 2. *The answer sets of a* HEX *program* Π *are the projections* $\mathbf{A}$ *of compatible sets* $\hat{\mathbf{A}}$ *where* $\{a \mid \mathbf{T}a \in \mathbf{A}\} \cap U = \emptyset$ *for every unfounded set* U *of* Π *wrt.* $\mathbf{A}$.

Example 6 *(Example 5 cont'd).* Recall that the assignment $\mathbf{A}_2 = \{\mathbf{T}p(a)\}$ is a compatible set but not an answer set of Π. In fact $U = \{p(a)\}$ is an unfounded set for Π wrt. $\mathbf{A}_2$: the condition (ii) is verified for the single rule $r = p(a) \leftarrow \&id[p](a)$, since $\&id[p](a)$ evaluates to false with respect to $\mathbf{A}_2\dot{\cup}\neg.U = \{\mathbf{F}p(a)\}$. Since $\{a \mid \mathbf{T}a \in \mathbf{A}_2\} \cap U = \{p(a)\} \neq \emptyset$, by Proposition 2, $\mathbf{A}_2$ is not answer set.

3. Pruning the external minimality check

Since an answer set $\hat{\mathbf{A}}$ of a guessing program $\hat{\Pi}$ must be a minimal model of the FLP-reduct $f\hat{\Pi}^{\hat{\mathbf{A}}}$, an e-minimality check is under certain conditions redundant. The criterion in [24] for deciding its necessity relies on an atom dependency graph induced by the HEX program. Informally, an e-minimality check is only needed for programs that allow cyclic support via external atoms, which can be checked efficiently. For instance, the program $\Pi_1 = \{\, p(a) \leftarrow \&id[p](a) \,\}$ allows cyclic support for the atom $p(a)$ via $\&id[p](a)$, while this is not the case for $\Pi_2 = \{\, p(a) \leftarrow \&id[q](a);\ q(a) \leftarrow r(a);\ r(a) \leftarrow q(a) \,\}$, where the truth value of $\&id[q](a)$ is independent of the value of p. If cyclic support via external atoms can be ruled out as for Π_2, the e-minimality check can be skipped for a program, potentially avoiding to invest many resources into a redundant check. Note, however, that a minimality check is still needed for computing the answer sets of $\hat{\Pi}$.

In this section, we introduce a new technique for skipping the e-minimality check wrt. a wider class of programs than previous approaches. More precisely, given Π, we present a new sufficient (i.e., sound) criterion for deciding the following *faithful guessing program* (*FGP*) property. Given an assignment $\hat{\mathbf{A}}$, let $\hat{\mathbf{A}}|_\Pi$ denote its restriction to the predicates of Π.

(FGP) For every compatible set $\hat{\mathbf{A}}$ for $\hat{\Pi}$, its restriction $\mathbf{A} = \mathbf{A}|_\Pi$ to the vocabulary of Π is an answer set of Π.

Like the criterion in [24], the criterion presented in this section exploits that output values of external atoms often do not depend on the complete extensions of their input predicates. However, while Eiter et al. [24] introduced a syntactic criterion for identifying cases in which (FGP) holds, we use additional information concerning semantic dependencies between the inputs and outputs of external atoms.

We note that our new criterion, like the previous syntactic criterion in [24], is sufficient but not necessary for (FGP) to hold. This can be seen by examples, and abstractly motivated by complexity arguments: testing the projection property is Π_2^p-hard for polynomial-time decidable external atoms already in the propositional case, and thus has higher worst-case complexity than minimality checking, which is co-NP-complete in this setting; we refer to Section 4.1 for further discussion.

3.1. Dependency graph pruning

We start by defining so-called *io-dependencies* for ge-predicates,[1] which specify that certain outputs of external atoms only depend on specific argument values of their inputs. For instance, whether a city c is in the output of $\&closeWest[city](X)$ from Section 1 only depends on cities c' that are located close to the east of c. Hence, the truth value

[1] Recall from Section 2 that for an external atom $\&g[\vec{p}](\vec{c})$, the external predicate with ground input list $\&g[\vec{p}]$ is called a ge-predicate.

of &*closeWest*[*city*](*kobe*) clearly only depends on the atom *city*(*osaka*), and we want to encode that *kobe* as first output of &*closeWest*[*city*](*X*) only depends on the element *osaka* as the first argument of the first input predicate *city*.

Definition 4 *(IO-Dependency).* An *io-dependency* for a ge-predicate &g[$\vec{p}$] is a tuple $\delta = \langle i, j : J, k : e\rangle$ where $1 \leq i \leq ar_I(\&g)$, $1 \leq j \leq ar(p_i)$, $p_i \in \vec{p}$, $1 \leq k \leq ar_O(\&g)$, $J \subseteq \mathcal{C}$ and $e \in \mathcal{C}$. The set of all δ for &g[$\vec{p}$] is denoted by $dep(\&g[\vec{p}])$.

In the sequel, io-dependencies will be used to constrain the possible dependencies between inputs and outputs of external atoms &g[$\vec{p}$]($\vec{c}$). Intuitively, an io-dependency $\langle i, j : J, k : e\rangle$ states that if constant e occurs as the kth output of &g[$\vec{p}$]($\vec{c}$), then only those input predicates at position i are relevant for its evaluation where the jth argument matches some $e' \in J$.

Example 7 *(Example 2 cont'd).* In our example, the io-dependency $\delta = \langle 1,\ 1:\{osaka\},\ 1:kobe\rangle$ could be specified for &*closeTo*[*city*](*kobe*), which could be provided by the user or generated automatically.

In order to simplify notation, we shall also omit "i," "j:" or "k:" from an io-dependency $\langle i, j : J, k : e\rangle$ for &g[$\vec{p}$] if it is uniquely determined; i.e., $i = 1$ if there is a single input predicate ($\vec{p} = p_1$), $j = 1$ if p_i is a unary predicate, and $k = 1$ if the external predicate is unary. The io-dependency δ in Example 7 could thus be written, in most compact form, as $\delta = \langle\{osaka\}, kobe\rangle$.

Io-dependencies induce atom sets relevant for evaluating respective external atoms:

Definition 5 *(Compliant Atoms).* A ground ordinary atom $p_i(\vec{d})$, with $\vec{d} = d_1, ..., d_l$, is *compliant* with a set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies for a ground external atom &g[$\vec{p}$]($\vec{c}$) if $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D$ with $e = c_k$. The set of all atoms compliant with D for &g[$\vec{p}$]($\vec{c}$) is denoted by $comp(D, \&g[\vec{p}](\vec{c}))$.

Example 8 *(Example 2, cont'd).* For our neighboring cities example we have $comp(\{\delta\}, \&closeTo[city](kobe))$=$\{city(osaka)\}$ and likewise $comp(\{\delta\}, \&closeWest[city](kobe)) = \{city(osaka)\}$.

The semantics of external atoms is related to io-dependencies as follows.

Definition 6 *(Faithfulness).* A set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies is *faithful*, if for any assignments $\mathbf{A}, \mathbf{A}'$ and ground external atom &g[$\vec{p}$]($\vec{c}$), either $\mathbf{A}(p_i(\vec{d})) \neq \mathbf{A}'(p_i(\vec{d}))$ for some $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ or $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.

Thus, io-dependencies $D \subseteq dep(\&g[\vec{p}])$ constrain the set of atoms that potentially impact the evaluation of &g[$\vec{p}$]($\vec{c}$), i.e. if D is faithful, changing only truth values of atoms $p_i(\vec{d})$ that are not in $comp(D, \&g[\vec{p}](\vec{c}))$ has no effect on the value of &g[$\vec{p}$]($\vec{c}$).

Example 9 *(Example 8, cont'd).* The io-dependency $\delta = \langle\{osaka\}, kobe\rangle$ from above is faithful wrt. &*closeTo*[*city*](*kobe*). Indeed, since $comp(\{\delta\}, \&closeTo[city](kobe)) = \{city(osaka)\}$ and whenever assignments $\mathbf{A}$ and $\mathbf{A}'$ agree on $city(osaka)$, i.e., whenever $\mathbf{A}(city(osaka)) = \mathbf{A}'(city(osaka))$, then $f_{\&closeTo}(\mathbf{A}, city, kobe) = f_{\&g}(\mathbf{A}', city, kobe)$ as *osaka* is the only city close to *kobe* according to the semantics as defined in Example 2; and similarly δ is faithful wrt. &*closeWest*[*city*](*kobe*).

In the following, we denote by $D(\&g[\vec{p}]) \subseteq dep(\&g[\vec{p}])$ a set of io-dependencies specified for &g[$\vec{p}$]. By default, we assume that $D(\&g[\vec{p}])$ is empty, but it can be utilized to supply additional dependency information. To ensure the correctness of an algorithm that skips e-minimality checks based on $D(\&g[\vec{p}])$, it is important that $D(\&g[\vec{p}])$ is faithful. In the following, we assume that the given sets of io-dependencies are faithful. In practice, this has to be ensured either by the user based on additional information regarding the input-output behavior of external atoms; or by an automatic check, which is tractable only in certain cases. The computational complexity of checking faithfulness is studied in detail in Section 4. Simultaneously, the goal is to approximate the real dependencies between atoms as close as possible for maximal performance gains. Note that while an extensional specification of $D(\&g[\vec{p}])$ might be very verbose, io-dependencies can often also be specified more concisely in an intensional manner, as in the following example.

Example 10. Consider &*setDiff*[*dom*, *set*](*c*), which is true for $c \in \mathcal{C}$ and assignment $\mathbf{A}$ iff $\{\mathbf{T}dom(c), \mathbf{F}set(c)\} \subseteq \mathbf{A}$. Thus, the presence of an output value c only depends on atoms with predicate *dom* or *set* that have c as first argument. Hence, $D(\&setDiff[dom, set]) = \{\langle 1, 1{:}\{c\}, 1{:}c\rangle, \langle 2, 1{:}\{c\}, 1{:}c\rangle \mid c \in \mathcal{C}\}$ is faithful.

We now introduce a notion of atom dependency in HEX programs that accounts for io-dependencies and generalizes the corresponding notion from [24].

Definition 7 *(Atom Dependency).* Given a ground HEX program Π, a set $D(\&g[\vec{p}])$ for each &g[$\vec{p}$] in Π, and ordinary ground atoms $p(\vec{d})$ and $q(\vec{e})$, we say

Fig. 1. Full and pruned dependency graph for Π from Example 11 (all arrows are "$\rightarrow_e$").

- $q(\vec{e})$ *depends on* $p(\vec{d})$, denoted $q(\vec{e}) \rightarrow_d p(\vec{d})$, if for some rule $r \in \Pi$ it holds that $q(\vec{e}) \in H(r)$ and $p(\vec{d}) \in B^+(r)$; and
- $q(\vec{e})$ *depends externally on* $p(\vec{d})$, denoted $q(\vec{e}) \rightarrow_e p(\vec{d})$, if for some rule $r \in \Pi$, some external atom $\&g[\vec{p}](\vec{c}) \in B^+(r) \cup B^-(r)$ with $p \in \vec{p}$ exists such that $q(\vec{e}) \in H(r)$ and $p(d) \in comp(D(\&g[\vec{p}]), \&g[\vec{p}](\vec{c}))$.

Note that Definition 7 generalizes the corresponding one from [24] in that an external dependency is only added if the specified io-dependencies are satisfied. The definitions coincide if $D(\&g[\vec{p}]) = \emptyset$ for all ge-predicates $\&g[\vec{p}]$ in Π.

Example 11. Consider $\&suc[node](n)$, which evaluates to true wrt. an assignment $\mathbf{A}$ and an external directed graph $\mathcal{G} = (V, E)$ iff $n' \rightarrow n \in E$ for some node n' such that $\mathbf{T}node(n) \in \mathbf{A}$. It is utilized in the following HEX program Π:

$$node(a).\ node(X) \leftarrow \&suc[node](X).$$

Intuitively, the program computes all nodes reachable from node a via the edges in $\mathcal{G}$. If the external graph has nodes $V = \{a, b, c, d\}$ and directed edges $E = \{a \rightarrow b,\ a \rightarrow c, c \rightarrow d, e \rightarrow d\}$, the grounding of Π produced by the grounding algorithm of the HEX program solver DLVHEX contains the following rules (omitting facts):

$$node(b) \leftarrow \&suc[node](b).\ node(c) \leftarrow \&suc[node](c).\ node(d) \leftarrow \&suc[node](d).$$

Without specifying io-dependencies for $\&suc[node]$, it holds, e.g., that $node(a) \rightarrow_e node(b)$ and $node(b) \rightarrow_e node(a)$. However, we can specify $D(\&suc[node]) = \left\{\langle\{c_1 \mid c_1 \rightarrow c_2 \in E\}, c_2\rangle \mid c_2 \in \mathcal{C}\right\}$, exploiting that the presence of output nodes only depends on input nodes to which they are successors. In this case, $node(a) \rightarrow_e node(b)$ does not hold according to Definition 7 as $b \rightarrow a \notin E$.

The following lemma states that external dependencies according to Definition 7 still cover all atoms relevant for deciding the truth value of an external atom, which follows directly from the definition of faithful io-dependencies.

Lemma 1. *Let Π be a HEX program, let r be a rule in Π, and let $\mathbf{A}$ and $\mathbf{A}'$ be two assignments. If $a \in H(r)$ and $\&g[\vec{q}](\vec{c}) \in B^+(r) \cup B^-(r)$, then $\mathbf{A} \models \&g[\vec{q}](\vec{c})$ iff $\mathbf{A}' \models \&g[\vec{q}](\vec{c})$ given that $\mathbf{A}(q(\vec{e})) = \mathbf{A}'(q(\vec{e}))$ for all atoms $q(\vec{e})$ where $a \rightarrow_e q(\vec{e})$.*

Proof. Let Π be a HEX program, let r be a rule in Π, and let $\mathbf{A}$ and $\mathbf{A}'$ be two assignments. Furthermore, let $a \in H(r)$ and $\&g[\vec{q}](\vec{c}) \in B^+(r) \cup B^-(r)$, and suppose that $\mathbf{A}(q(\vec{e})) = \mathbf{A}'(q(\vec{e}))$ holds for all atoms q where $a \rightarrow_e q(\vec{e})$. According to Definition 7, for all $q(\vec{e})$ where $a \rightarrow_e q(\vec{e})$ it holds that $q(\vec{e}) \in comp(D(\&g[\vec{q}]), \&g[\vec{q}](\vec{c}))$. Furthermore, it holds that $a \rightarrow_e q(\vec{e})$ for every $q(\vec{e}) \in comp(D(\&g[\vec{q}]), \&g[\vec{q}](\vec{c}))$ because $q(\vec{e}) \in comp(D(\&g[\vec{q}]), \&g[\vec{q}](\vec{c}))$ is only possible if $q \in \vec{q}$. Hence, $\mathbf{A} \models \&g[\vec{q}](\vec{c})$ iff $\mathbf{A}' \models \&g[\vec{q}](\vec{c})$ follows by Definition 6, and since we assume all sets of io-dependencies to be faithful wrt. the given HEX program. □

We are now ready to introduce the atom dependency graph for a given program Π. From this graph, a property of Π can be derived which is subsequently employed to decide the necessity of the e-minimality check wrt. Π.

Definition 8 *(Dependency Graph).* Given a ground HEX program Π, the *dependency graph* $\mathcal{G}_\Pi^{dep} = (V, E)$ has the vertices $V = A(\Pi)$ and directed edges $E = \rightarrow_d \cup \rightarrow_e$; Π has an *e-cycle*, if $\mathcal{G}_\Pi^{dep}$ has a cycle with an edge $\rightarrow_e$.

By specifying io-dependencies, the external dependencies in the dependency graph of a HEX program can be pruned.

Example 12 *(Example 11 cont'd).* Fig. 1 shows the dependency graphs for Π from Example 11, with and without specified io-dependencies. The full dependency graph has an e-cycle, but the pruned graph does not.

In the remainder of this subsection, our goal is to show that the e-minimality check can be skipped for all those HEX programs that do not have an e-cycle. Notably, while the inverse of $\rightarrow_d$ was additionally included in $\mathcal{G}_\Pi^{dep}$ by Eiter et al. [24], we improve their results by showing that our more general definition suffices.

For proving the main result of this section, i.e. correctness of our new decision criterion, we first need to introduce two lemmas from [24], which are used subsequently in the proof of Theorem 1; and the concept of a *cut*, which intuitively

represents a set of atoms that do not belong to the *core* of an unfounded set. The following definition is a generalization of Definition 13 in [24]:

Definition 9 *(Cut).* Let U be an unfounded set of Π wrt. $\mathbf{A}$. A set of atoms $C \subseteq \{a \mid \mathbf{T}a \in U\}$ is a *cut* of $\mathcal{G}_\Pi^{dep}$, if

(i) $b \not\rightarrow_e a$, for all $a \in C$ and $b \in \{a \mid \mathbf{T}a \in U\}$ (C has no incoming e-edge from U),
(ii) $b \not\rightarrow_d a$, for all $a \in C$ and $b \in \{a \mid \mathbf{T}a \in U\} \setminus C$ (there are no ordinary edges $\rightarrow_d$ from $\{a \mid \mathbf{T}a \in U\} \setminus C$ to C).

Next, we establish two lemmas which will help proving the main result of this section. They correspond to Lemmas 18 and 19 in [24], which are lifted to our more general setting. We follow in the proofs the line of argument in [24], where the difference is that our generalized definitions of atom dependencies, the dependency graph $\mathcal{G}_\Pi^{dep}$, and cuts are used and that the correctness of external evaluations is ensured by Lemma 1.

The first lemma states that atoms in a cut of an unfounded set can be removed while the resulting set still constitutes an unfounded set.

Lemma 2. *Let U be an unfounded set of Π wrt. $\mathbf{A}$, and let C be a cut. Then, $Y = \{a \mid \mathbf{T}a \in U\} \setminus C$ is an unfounded set of Π wrt. $\mathbf{A}$.*

Proof. If $Y = \emptyset$, then the result holds trivially. Otherwise, let $r \in \Pi$ with $H(r) \cap Y \neq \emptyset$. Observe that $H(r) \cap U \neq \emptyset$ because $U \supseteq Y$. Since U is an unfounded set of Π wrt. $\mathbf{A}$, according to Definition 3, either

(i) $\mathbf{A} \not\models b$ for some $b \in B(r)$; or
(ii) $\mathbf{A} \dot\cup \neg.U \not\models b$ for some $b \in B(r)$; or
(iii) $\mathbf{A} \models h$ for some $h \in H(r) \setminus U$.

In case (i), the condition also holds wrt. Y. In case (ii), let $a \in H(r)$ such that $a \in Y$, and $b \in B(r)$ such that $\mathbf{A} \dot\cup \neg.U \not\models b$. We make a case distinction: either b is an ordinary literal or an external one.

If b is an ordinary default-negated atom $\mathrm{not}\, c$, then $\mathbf{A} \dot\cup \neg.U \not\models b$ implies $\mathbf{T}c \in \mathbf{A}$ and $c \notin U$, and therefore also $\mathbf{A} \dot\cup \neg.Y \not\models b$. So assume b is an ordinary atom. If $b \notin U$ then $\mathbf{A} \not\models b$ and case (i) applies, so assume $b \in U$. Because $a \in H(r)$ and $b \in B(r)$, we have $a \rightarrow_d b$. By assumption $a \in Y$, and therefore $b \in Y$ because there are no ordinary edges $\rightarrow_d$ from Y to C according to Definition 9. Hence $\mathbf{A} \dot\cup \neg.Y \not\models b$.

If b is an external literal, then there is no $q \in U$ with $a \rightarrow_e q$ and $q \notin Y$. Otherwise, this would imply $q \in C$ and C would have an incoming e-edge, which contradicts the assumption that C is a cut. Hence, for all $q \in U$ with $a \rightarrow_e q$, also $q \in Y$, and therefore the truth value of b under $\mathbf{A} \dot\cup \neg.U$ and $\mathbf{A} \dot\cup \neg.Y$ is the same, according to Lemma 1. Hence $\mathbf{A} \dot\cup \neg.Y \not\models b$.

In case (iii), then also $\mathbf{A} \models h$ for some $h \in H(r) \setminus Y$ because $Y \subseteq U$ and therefore $H(r) \setminus Y \supseteq H(r) \setminus U$. □

The second lemma states that any unfounded set U of a HEX program Π can be detected during the evaluation of the guessing program $\hat\Pi$ if no input to some external atom is contained in U.

Lemma 3. *Let U be an unfounded set of Π wrt. $\mathbf{A}$. If there are no $x, y \in \{a \mid \mathbf{T}a \in U\}$ such that $x \rightarrow_e y$, then U is an unfounded set of $\hat\Pi$ wrt. $\hat{\mathbf{A}}$.*

Proof. If $U = \emptyset$, then the result holds trivially. Otherwise, suppose $\hat r \in \hat\Pi$ and $H(\hat r) \cap U \neq \emptyset$. Let $a \in H(\hat r) \cap U$. Observe that $\hat r$ cannot be an external atom guessing rule because U contains only ordinary atoms. We show that one of the conditions in Definition 3 holds for $\hat r$ wrt. $\hat{\mathbf{A}}$.

Because $\hat r$ is no external atom guessing rule, there is a corresponding rule $r \in \Pi$ containing external atoms in place of replacement atoms. Because U is an unfounded set of Π and $H(r) = H(\hat r)$, according to Definition 3, either:

(i) $\mathbf{A} \not\models b$ for some $b \in B(r)$; or
(ii) $\mathbf{A} \dot\cup \neg.U \not\models b$ for some $b \in B(r)$; or
(iii) $\mathbf{A} \models h$ for some $h \in H(r) \setminus U$.

In case (i), let $b \in B(r)$ such that $\mathbf{A} \not\models b$ and $\hat b$ the corresponding literal in $B(\hat b)$ (which is the same if b is ordinary and the corresponding replacement literal if b is external). Then also $\hat{\mathbf{A}} \not\models \hat b$ because $\hat{\mathbf{A}}$ is compatible.

In case (ii), we make a case distinction: either b is ordinary or external.

If b is ordinary, then $b \in B(\hat r)$ and $\mathbf{A} \dot\cup \neg.U \not\models b$ holds because $\mathbf{A}$ and $\hat{\mathbf{A}}$ are equivalent for ordinary atoms. If b is an external atom $\&g[\vec q](\vec c)$ or a default-negated external atom $\mathrm{not}\,\&g[\vec q](\vec c)$, then $q(\vec e) \notin U$ for all atoms $q(\vec e)$ where $a \rightarrow_e q(\vec e)$; otherwise we would have a contradiction to our assumption that U has no internal e-edges. Hence, we have that $\mathbf{A} \dot\cup \neg.U(q(\vec e)) = \mathbf{A}(q(\vec e))$ for all atoms $q(\vec e)$ where $a \rightarrow_e q(\vec e)$. But then $\mathbf{A} \dot\cup \neg.U \not\models b$ implies $\mathbf{A} \not\models b$ according to Lemma 1. Therefore we can apply case (i).

In case (iii), also $\hat{\mathbf{A}} \models h$ for some $h \in H(\hat{r}) \setminus U$ because $H(r) = H(\hat{r})$ contains only ordinary atoms and $\mathbf{A}$ is equivalent to $\hat{\mathbf{A}}$ for ordinary atoms. □

The following theorem represents the main result of this section and differs from the previous result for e-minimality check skipping by [24] in that it is based on our generalized definition of external dependencies. Consequently, it can be applied to a larger class of HEX programs.

Theorem 1. *If a ground* HEX *program* Π *contains no e-cycle, then for every compatible set* $\hat{\mathbf{A}}$ *for* $\hat{\Pi}$ *it holds that its restriction* $\mathbf{A} = \hat{\mathbf{A}}|_\Pi$ *is an answer set of* Π.

Proof. The core of the following proof mirrors the proof for Theorem 20 in [24], but it has been adapted to our more general definition of the atom dependency graph $\mathcal{G}_\Pi^{dep}$.

Since the answer sets of a HEX program Π are exactly the projections $\mathbf{A}$ of compatible sets $\hat{\mathbf{A}}$ where $\{a \mid \mathbf{T}a \in \mathbf{A} \cap U\} = \emptyset$ for every unfounded set U of Π wrt. $\mathbf{A}$ (cf. Section 2), we can prove the theorem by showing that the following statement holds:

If a ground HEX program Π contains no e-cycle, and no unfounded set U of $\hat{\Pi}$ wrt. an assignment $\hat{\mathbf{A}}$ such that $\{a \mid \mathbf{T}a \in \hat{\mathbf{A}} \cap U\} \neq \emptyset$ exists, then no unfounded set U' of Π wrt. $\mathbf{A}$ such that $\{a \mid \mathbf{T}a \in \mathbf{A} \cap U'\} \neq \emptyset$ exists, where $\mathbf{A}$ is the projection of $\hat{\mathbf{A}}$.

We prove the contrapositive. Let Π be a HEX program that contains no e-cycle, and let $\mathbf{A}$ be the projection of an assignment $\hat{\mathbf{A}}$ s.t. an unfounded set U' for Π wrt. $\mathbf{A}$ s.t. $\{a \mid \mathbf{T}a \in \mathbf{A} \cap U'\} \neq \emptyset$ exists. Since U' is an unfounded set for Π wrt. $\mathbf{A}$ s.t. $\{a \mid \mathbf{T}a \in \mathbf{A} \cap U'\} \neq \emptyset$, the set $U'' = \{\mathbf{T}a \mid \mathbf{T}a \in \mathbf{A} \cap U'\}$ is also a nonempty unfounded set for Π wrt. $\mathbf{A}$. The previous holds because no atom in $\{a \mid \mathbf{T}a \in \mathbf{A}\} \setminus \{a \mid \mathbf{T}a \in U'\}$ is true under $\mathbf{A}$ anyway and hence, conditions (i) to (iii) from the definition of unfounded sets must be satisfied wrt. $\{a \mid \mathbf{T}a \in \mathbf{A} \cap U'\}$ as well. We need to show that an unfounded set U of $\hat{\Pi}$ wrt. $\hat{\mathbf{A}}$ such that $\{a \mid \mathbf{T}a \in \hat{\mathbf{A}} \cap U\} \neq \emptyset$ exists as well.

Let $\leftarrow_d$ be the inverse of $\rightarrow_d$. We define the *reachable set* $R(a)$ from some atom a as

$$R(a) = \{b \mid (a, b) \in \leftarrow_d^*\},$$

i.e. the set of atoms $b \in \{a \mid \mathbf{T}a \in U\}$ reachable from a using edges from $\leftarrow_d$ only but no e-edges.

We first assume that $\{a \mid \mathbf{T}a \in U''\}$ contains at least one e-edge, i.e. there are $x, y \in \{a \mid \mathbf{T}a \in U''\}$ such that $x \rightarrow_e y$. Now we show that there is a $u \in \{a \mid \mathbf{T}a \in U''\}$ with outgoing e-edge (i.e. $u \rightarrow_e v$ for some $v \in \{a \mid \mathbf{T}a \in U''\}$), but such that $R(u)$ has no incoming e-edges (i.e. for all $v \in R(u)$ and $b \in \{a \mid \mathbf{T}a \in U''\}$, $b \not\rightarrow_e v$ holds). Suppose to the contrary that for all a with outgoing e-edges, the reachable set $R(a)$ has an incoming e-edge. We now construct an e-cycle under $\rightarrow_d \cup \rightarrow_e$, which contradicts our assumption. Start with an arbitrary node with an outgoing e-edge $c_0 \in \{a \mid \mathbf{T}a \in U''\}$ and let p_0 be the (possibly empty) path (under $\leftarrow_d$) from c_0 to the node $d_0 \in R(c_0)$ such that d_0 has an incoming e-edge, i.e. there is a c_1 such that $c_1 \rightarrow_e d_0$; note that $c_1 \notin R(c_0)$.[2] By assumption, also some node d_1 in $R(c_1)$ has an incoming e-edge (from some node $c_2 \notin R(c_1)$). Let p_1 be the path from c_1 to d_1, etc. By iteration we can construct the concatenation of the paths $p_0, (d_0, c_1), p_1, (d_1, c_2), p_2, \ldots, p_i, (d_i, c_{i+1}), \ldots$, where the p_i from c_i to d_i are the paths within reachable sets, and the (d_i, c_{i+1}) are the e-edges between reachable sets. However, as $\{a \mid \mathbf{T}a \in U''\}$ is finite, some nodes on this path must be equal, i.e., a prefix of the constructed sequence represents an e-cycle (in reverse order).

This proves that u is a node with outgoing e-edge but such that $R(u)$ has no incoming e-edges. We next show that $R(u)$ is a cut. Condition (i) of Definition 9 is immediately satisfied by definition of u. Condition (ii) is shown as follows. Let $u' \in R(u)$ and $v' \in \{a \mid \mathbf{T}a \in U''\} \setminus R(u)$. We have to show that $v' \not\rightarrow_d u'$. Suppose, towards a contradiction, that $v' \rightarrow_d u'$. Because of $u' \in R(u)$, there is a path from u to u' under $\leftarrow_d$. But if $v' \rightarrow_d u'$, then there would also be a path from u to v' under $\leftarrow_d$ and v' would be in $R(u)$, a contradiction.

Therefore, $R(u)$ is a cut of U'', and by Lemma 2, it follows that $\{a \mid \mathbf{T}a \in U''\} \setminus R(u)$ is an unfounded set. Observe that $\{a \mid \mathbf{T}a \in U''\} \setminus R(u)$ contains one e-edge less than $\{a \mid \mathbf{T}a \in U''\}$ because u has an outgoing e-edge. Further observe that $\{a \mid \mathbf{T}a \in U''\} \setminus R(u) \neq \emptyset$ because there is a $w \in \{a \mid \mathbf{T}a \in U''\}$ such that $u \rightarrow_e w$ but $w \notin R(u)$. By iterating this argument, the number of e-edges in the unfounded set can be reduced to zero in a nonempty core. Eventually, Lemma 3 applies, proving that the remaining set is an unfounded set of $\hat{\Pi}$. Hence, we infer that there is a nonempty unfounded set U of $\hat{\Pi}$ wrt. $\hat{\mathbf{A}}$. Moreover, we have that $\{a \mid \mathbf{T}a \in U\} \subseteq \{a \mid \mathbf{T}a \in U''\}$ by construction of U and since $\{a \mid \mathbf{T}a \in U''\} \subseteq \{a \mid \mathbf{T}a \in \mathbf{A}\}$, we also have that $\{a \mid \mathbf{T}a \in \hat{\mathbf{A}} \cap U\} \neq \emptyset$. □

Example 13 *(Example 12 cont'd).* Since the pruned dependency graph from Example 11 does not have an e-cycle, Π does not require e-minimality checks (cf. Theorem 1). Note that this can only be detected using the pruned graph.

As a result, we obtain a flexible means for increasing the efficiency of evaluating a class of HEX programs where the e-minimality check is performed due to an overapproximation of the real dependencies between atoms.

[2] Whenever $x \rightarrow_e y$ for $x, y \in \{a \mid \mathbf{T}a \in U''\}$, then there is no path from x to y under $\leftarrow_d$, because otherwise we would have an e-cycle under $\rightarrow_d \cup \rightarrow_e$.

3.2. Semantic properties of faithful io-dependencies

Next, we start by studying semantic properties of io-dependencies independent from a HEX program, before we consider properties of relativized io-dependencies in the next section. In particular, we focus on properties important for checking faithfulness of sets of io-dependencies, as well as for generating and optimizing them.

Informally, given two sets of io-dependencies $D_1, D_2 \subseteq dep(\&g[\vec{p}])$, D_1 is better than D_2 if it induces less compliant atoms. We thus say that D_1 *tightens* D_2, denoted $D_1 \leq D_2$, if $comp(D_1, \&g[\vec{p}](\vec{c})) \subseteq comp(D_2, \&g[\vec{p}](\vec{c}))$ holds for all tuples $\vec{c}$. We call D_1 *tight* if no D_2 strictly tightens D_1, i.e., no D_2 exists such that $D_2 \leq D_1$ but $D_1 \not\leq D_2$. Furthermore, we call D_1 and D_2 *equally tight*, denoted $D_1 \equiv D_2$, if $D_1 \leq D_2$ and $D_2 \leq D_1$. We then have:

Proposition 3 *(Faithfulness Propagation). Suppose sets $D_1, D_2 \subseteq dep(\&g[\vec{p}])$ of io-dependencies are such that $D_1 \leq D_2$. If D_1 is faithful, then D_2 is also faithful.*

Proof. Towards a contradiction, suppose that $D_1 \leq D_2$ and D_1 is faithful, but D_2 is not. Then there exist assignments $\mathbf{A}, \mathbf{A}'$ and a tuple $\vec{c}$ such that $\mathbf{A}$ and $\mathbf{A}'$ coincide on $comp(D_2, \&g[\vec{p}](\vec{c}))$ but $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) \neq f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$. Because it holds that $comp(D_1, \&g[\vec{p}](\vec{c})) \subseteq comp(D_2, \&g[\vec{p}](\vec{c}))$, it follows that $\mathbf{A}$ and $\mathbf{A}'$ coincide also on $comp(D_1, \&g[\vec{p}](\vec{c}))$; but this contradicts that D_1 is faithful. □

As a consequence, faithfulness is anti-monotonic wrt. set-inclusion, and it is monotonic wrt. adding subsumed io-dependencies, where $\delta = \langle i, j : J, k : e\rangle$ *subsumes* $\delta' = \langle i, j : J', k : e\rangle$, if $J \subseteq J'$ holds.

Corollary 1. *If a set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies is faithful, then (i) each $D' \subseteq D$ is faithful and (ii) each $D' = D \cup D''$ where each $\delta'' \in D''$ is subsumed by some $\delta \in D$ is faithful.*

Proof. As for (i), that $D' \subseteq D$ implies $D' \leq D$: each atom $p(\vec{d})$ that is compliant for D' is also compliant for D, as $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D'$ with $e = c_k$ trivially implies that $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D$ with $e = c_k$. For (ii), we likewise conclude for $D' = D \cup D''$ where each $\delta'' \in D''$ is subsumed by some $\delta \in D$, that $D' \leq D$ as $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D'$ with $e = c_k$ implies that $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D$ with $e = c_k$. As D is faithful, (i) and (ii) follow thus from Proposition 3. □

Consequently, we can tighten a faithful set D by sequentially dropping constants c from io-dependencies $\delta = \langle i, j : J, k : e\rangle$ in D, i.e., check whether $D \cup \delta'$ for $\delta' = \langle i, j : J \setminus \{c\}, k : e\rangle$ is faithful and if so, replace D with $(D \setminus \{\delta\}) \cup \{\delta'\}$.

We can simplify D by exploiting the following equivalences; let $\delta^*(i, j, k{:}e) = \langle i, j : \mathcal{C}, k : e\rangle$ for any possible i, j, and $k : e$.

Proposition 4. *For a set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies and an io-dependency $\langle i, j : J, k : e\rangle \in dep(\&g[\vec{p}])$, we have (i) $D \equiv D \cup \{\delta^*(i, j, k{:}e)\} \equiv D \setminus \{\delta^*(i, j, k{:}e)\}$, and (ii) for any $\delta = \langle i, j : J, k : e\rangle$, $\delta' = \langle i, j : J', k : e\rangle \in D$ that $D \equiv D \cup \{\langle i, j : J \cap J', k : e\rangle\}$.*

Proof. Let $D \subseteq dep(\&g[\vec{p}])$ for a ge-predicate $\&g[\vec{p}]$, and let $\langle i, j : J, k : e\rangle \in dep(\&g[\vec{p}])$.

(i) First, we have that $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ if and only if $p_i(\vec{d}) \in comp(D \cup \{\delta^*(i, j, k{:}e)\}, \&g[\vec{p}](\vec{c}))$ because $d_j \notin J'$ for some $\langle i, j : J', k : e\rangle \in D$ implies that $d_j \notin J''$ for some $\langle i, j : J'', k : e\rangle \in D \cup \{\delta^*(i, j, k{:}c)\}$, and it holds trivially that $d_j \in \mathcal{C}$ for every $d_j \in J'$ and every $\langle i, j : J', k : e\rangle \in D \cup \{\delta^*(i, j, k{:}e)\}$.
We can also show that $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ if and only if $p_i(\vec{d}) \in comp(D \setminus \{\delta^*(i, j, k{:}e)\}, \&g[\vec{p}](\vec{c}))$. First, $d_j \notin J'$ for some $\langle i, j : J', k : e\rangle \in D$ implies that $d_j \notin J'$ for some $\langle i, j : J', k : e\rangle \in D \setminus \{\delta^*(i, j, k{:}c)\}$ because $d_j \in \mathcal{C}$. Moreover, if $d_j \in J'$ for every $\langle i, j : J', k : e\rangle \in D$, then $d_j \in J''$ for every $\langle i, j : J'', k : e\rangle \in D \setminus \{\delta^*(i, j, k{:}c)\}$ since $D \setminus \{\delta^*(i, j, k{:}c)\} \subseteq D$.

(ii) Next, let $\delta = \langle i, j{:}J, k{:}e\rangle$, $\delta' = \langle i, j{:}J', k{:}e\rangle \in D$ be arbitrary io-dependencies in D. We prove that $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ if and only if $p_i(\vec{d}) \in comp(D \cup \{\langle i, j{:}J \cap J', k{:}e\rangle\}, \&g[\vec{p}](\vec{c}))$. Let $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ be a compliant atom. Then, $d_j \in J \cap J'$ because $d_j \in J$ and $d_j \in J'$, which shows that $p_i(\vec{d}) \in comp(D \cup \{\langle i, j{:}J \cap J', k{:}e\rangle\}, \&g[\vec{p}](\vec{c}))$. Finally, let $p_i(\vec{d})$ be a ground ordinary atom s.t. $p_i(\vec{d}) \notin comp(D, \&g[\vec{p}](\vec{c}))$. But then $p_i(\vec{d}) \notin comp(D \cup \{\langle i, j{:}J \cap J', k{:}e\rangle\}, \&g[\vec{p}](\vec{c}))$ as $d_j \notin J''$ for some $\langle i, j : J'', k : e\rangle \in D$. □

That is, $\delta^*(i, j, k{:}e)$ is like a tautology, and we can replace all dependencies for i, j and $k : e$ in D by one which contains the intersection of all their J-sets. We thus can *normalize* D into $nf(D)$ such that for each i, j, and $k : e$ exactly one io-dependency occurs, and then start tightening. We then obtain:

Proposition 5 *(Faithful Tightening). Given a faithful set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies, exhaustive tightening of $nf(D)$ results in a tight faithful D'.*

Proof. First, D' is faithful because faithfulness is checked each time before an io-dependency $\delta = \langle i, j : J, k : e\rangle$ is replaced by $\delta' = \langle i, j : J \setminus \{c\}, k : e\rangle$ during tightening. We need to show that D' is also tight.

Towards a contradiction, suppose that D' is exhaustively tightened but that it is not tight, i.e. that there is a set $D'' \subseteq dep(\&g[\vec{p}])$ s.t. $D'' \leq D'$ and $D' \not\leq D''$. This implies that there is some $p_i \in \vec{p}$ s.t. $p_i(\vec{d}) \in comp(D', \&g[\vec{p}](\vec{c}))$ and $p_i(\vec{d}) \notin comp(D'', \&g[\vec{p}](\vec{c}))$ for some output $\vec{c}$ of $\&g[\vec{p}]$. Thus, for some $1 \leq j \leq ar(p)$ we have that $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D'$ with $e = c_k$ but $d_j \notin J$ for some $\langle i, j : J, k : e\rangle \in D''$ with $e = c_k$, according to the definition of compliant atoms. But then, $D' \cup \{\delta'\}$ for $\delta' = \langle i, j : J \setminus \{d_j\}, k : e\rangle$ with $e = c_k$, where $\langle i, j : J, k : e\rangle \in D$ with $e = c_k$, is also faithful due to Proposition 3 because it holds that $D'' \leq D' \cup \{\delta'\}$, and D'' is faithful. This means that D' is not exhaustively tightened, which contradicts our assumption. □

The set $D = \emptyset$ is trivially faithful, and $nf(\emptyset)$ consists of all $\delta^*(i, j, k{:}c)$; thus even without user input, a tight faithful set D' for $\&g[\vec{p}]$ is constructible. Moreover, semantically faithful sets of compliant atoms have the intersection property.

Proposition 6 *(Faithful Intersection). If sets $D_1, D_2 \subseteq dep(\&g[\vec{p}])$ of io-dependencies are faithful, then $D_1 \cup D_2$ is faithful, and for every tuple $\vec{c}$ of constants, $comp(D_1 \cup D_2, \&g[\vec{p}](\vec{c})) = comp(D_1, \&g[\vec{p}](\vec{c})) \cap comp(D_2, \&g[\vec{p}](\vec{c}))$.*

Proof. Let $\vec{c}$ be an arbitrary output tuple of $\&g[\vec{p}]$, let $D_1, D_2 \subseteq dep(\&g[\vec{p}])$, and let $C_D = comp(D, \&g[\vec{p}](\vec{c}))$ for any $D \subseteq dep(\&g[\vec{p}])$.

We begin by showing that $C_{D_1 \cup D_2} = C_{D_1} \cap C_{D_2}$. For a ground atom $p(\vec{d})$, with $\vec{d} = d_1, ..., d_l$, it holds that $p(\vec{d}) \in C_{D_1 \cup D_2}$ iff $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D_1 \cup D_2$ with $e = c_k$. The previous holds iff $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D_1$ and $d_j \in J'$ for all $\langle i, j : J', k : e\rangle \in D_2$ with $e = c_k$, which holds iff $p(\vec{d}) \in C_{D_1}$ and $p(\vec{d}) \in C_{D_2}$. This proves that $p(\vec{d}) \in C_{D_1} \cap C_{D_2}$ iff $p(\vec{d}) \in C_{D_1 \cup D_2}$.

Now, we prove that if D_1 and D_2 are faithful, then $D_1 \cup D_2$ is faithful. Recall that according to Definition 4, a set of io-dependencies $D \subseteq dep(\&g[\vec{p}])$ is faithful iff:

(*) for any assignments $\mathbf{A}, \mathbf{A}'$ and ground external atom $\&g[\vec{p}](\vec{c})$, either $\mathbf{A}(p_i(\vec{d})) \neq \mathbf{A}'(p_i(\vec{d}))$ for some $p_i(\vec{d}) \in C_D$ or $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.

Suppose that statement (*) holds wrt. C_{D_1} and C_{D_2}; we show that it also holds wrt. $C_{D_1 \cup D_2} = C_{D_1} \cap C_{d_2}$.

Towards a contradiction, suppose that $C_{D_1 \cup D_2}$ does not satisfy (*). Hence there exist assignments $\mathbf{A}, \mathbf{A}'$ such that $\mathbf{A}(p_i(\vec{d})) = \mathbf{A}'(p_i(\vec{d}))$ for all $p_i(\vec{d}) \in C_{D_1 \cup D_2}$ and $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) \neq f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.

Let $\mathbf{A}_1$ be the assignment such that

$$\mathbf{A}_1(p_i(\vec{d})) = \begin{cases} \mathbf{A}(p_i(\vec{d})) & \text{if } p_i(\vec{d}) \in C_{D_1}, \\ \mathbf{A}'(p_i(\vec{d})) & \text{if } p_i(\vec{d}) \in C_{D_2} \setminus C_{D_1}, \\ \mathbf{T} & \text{otherwise.} \end{cases}$$

Then, as $\mathbf{A}_1$ and $\mathbf{A}$ coincide on C_{D_1} and the latter satisfies (*), it follows $f_{\&g}(\mathbf{A}_1, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}, \vec{p}, \vec{c})$.

Let $\mathbf{A}'_1$ be similarly the assignment such that

$$\mathbf{A}'_1(p_i(\vec{d})) = \begin{cases} \mathbf{A}'(p_i(\vec{d})) & \text{if } p_i(\vec{d}) \in C_{D_2}, \\ \mathbf{A}(p_i(\vec{d})) & \text{if } p_i(\vec{d}) \in C_{D_1} \setminus C_{D_2}, \\ \mathbf{T} & \text{otherwise.} \end{cases}$$

Then, as $\mathbf{A}'_1$ and $\mathbf{A}'$ coincide on C_{D_2} and the latter satisfies (*), it follows $f_{\&g}(\mathbf{A}'_1, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$. As $\mathbf{A}$ and $\mathbf{A}'$ coincide on $C_{D_1 \cup D_2} = C_{D_1} \cap C_{D_2}$, by construction $\mathbf{A}_1 = \mathbf{A}'_1$, and it follows $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}_1, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}'_1, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$; however, this contradicts the assumption that $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) \neq f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$. □

As an immediate consequence of the intersection property, we obtain the following property. Let us call a set $D^\star \subseteq dep(\&g[p])$ of faithful io-dependencies for $\&g[p]$ *tightest*, if for each set $D \subseteq dep(\&g[p])$ of faithful io-dependencies it holds that $D^\star \leq D$.

Corollary 2. *For every ge-predicate $\&g[p]$ a tightest set $D^\star = D^\star(\&g[p]) \subseteq dep(\&g[p])$ of faithful io-dependencies exists.*

Example 14. For the external predicate $\&closeTo[city]$ from Example 2, we have $D^\star = \{\langle\{osaka\}, kobe\rangle, \langle\{kobe\}, osaka\rangle, \langle\{vienna\}, bratislava\rangle, \langle\{bratislava\}, vienna\rangle\}$, while for $\&closeWest[city]$ we have $D^\star = \{\langle\{osaka\}, kobe\rangle, \langle\{bratislava\}, vienna\rangle\}$. Furthermore, in Examples 11 and 10, the sets of specified io-dependencies $D(\&suc[node])$ and $D(\&setDiff[dom, set])$, respectively, are both tightest faithful sets of io-dependencies.

However, while semantically the tightest faithful set $D^\star$ is unique, syntactically different tight faithful sets may exist.

Example 15. Consider a binary ge-predicate $\&g[p]$ which is true for output (a, b) wrt. an assignment $\mathbf{A}$ iff $\mathbf{T}p(c) \in \mathbf{A}$, and false for all other output tuples. Then $D_1 = \{\langle 1, 1 : \{c\}, 1 : a\rangle\}$ and $D_2 = \{\langle 1, 1 : \{c\}, 2 : b\rangle\}$ are faithful for $\&g[p]$, and as easily seen both are tight.

3.3. Relativized io-dependencies

So far, the context of a given HEX program has not been exploited for specifying respective io-dependencies. However, without considering how dependencies in an external source may be affected by input parameters, all io-dependencies that may hold under any possible extension of input predicates must be respected. This is illustrated by the following example.

Example 16. Consider &*suc*[*edge*, *node*](*X*), where edges from *edge* are inserted into $\mathcal{G}$ before successor nodes are output. If it is unknown which edges can be added, io-dependencies must account for the complete graph (all edges), which is a maximal overapproximation. Now, consider the following HEX program.

$$edge(b,c) \vee n\_edge(b,c).\ \ node(a).\ \ node(b) \leftarrow \&suc[edge, node](b).$$

As $edge(b,c)$ is the only atom with a predicate *edge* that can potentially be true in the input of &*suc*[*edge*, *node*](*b*) in any answer set, it suffices to specify io-dependencies wrt. the graph $\mathcal{G}' = (V, E \cup \{b \to c\})$ to ensure e-minimality.

To account for the inputs to external sources that are possible in answer sets, we define faithfulness of io-dependencies relative to some set E of atoms, and based on this faithfulness wrt. a HEX program Π.

Definition 10 *(Core Envelope).* For any HEX program Π, we denote with $env(\Pi)$ the set of all ground atoms that are true in some answer set of Π.

Since answer sets of HEX programs are supported models, i.e., an atom a can only be true if some rule instance r' with a in the head fires, only assignments $\mathbf{A}$ are relevant as candidate answer sets of Π such that $\{a \mid \mathbf{T}a \in \mathbf{A}\} \subseteq env(\Pi)$ holds, i.e., each atom that is true in $\mathbf{A}$ is from $env(\Pi)$; the same applies for every assignment $\mathbf{A}' \leq \mathbf{A}$ that refutes that $\mathbf{A}$ is an answer set, i.e., $\mathbf{A}' \neq \mathbf{A}$ satisfying $grnd(\Pi)^{\mathbf{A}}$. We now define relativized faithfulness as follows.

Definition 11 *(Relativized Faithfulness).* A set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies is *faithful wrt. a set E of ground atoms*, if for any assignments $\mathbf{A}$, $\mathbf{A}'$ such that $\{a \mid \mathbf{T}a \in \mathbf{A} \cup \mathbf{A}'\} \subseteq E$, and for any output tuple $\vec{c}$ for $\&g[\vec{p}]$, either $\mathbf{A}(p_i(\vec{d})) \neq \mathbf{A}'(p_i(\vec{d}))$ for some atom $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ or $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$. Furthermore, $D \subseteq dep(\&g[\vec{p}])$ is *faithful wrt. a HEX program* Π, if D is faithful wrt. $E = env(\Pi)$.

Note that (unrestricted) faithfulness is obtained for E consisting of all ground atoms. We show that skipping e-minimality checks based on the relativized definition of faithful io-dependencies is still safe, provided the set E is large enough.

Proposition 7. *Theorem 1 still holds if the specified io-dependencies are faithful wrt. a set E of ground atoms such that $E \supseteq env(\Pi)$ for the given HEX program Π.*

Proof. The proof of Theorem 1 can be directly adopted because, in order to ensure that the projection $\mathbf{A}$ of a compatible set $\hat{\mathbf{A}}$ for a HEX program Π is a $\leq$-minimal model of $f\Pi^{\mathbf{A}}$, one only needs to consider assignments $\mathbf{A}'$ such that $\mathbf{A}' \leq \mathbf{A}$; and we have that $\mathbf{A}' \leq \{\mathbf{T}a \mid a \in env(\Pi)\}$ for the projection $\mathbf{A}' = \hat{\mathbf{A}}'|_{\Pi}$ of every compatible set $\hat{\mathbf{A}}'$ for Π. □

The properties of above can be adjusted to the relativized setting. Specifically, we obtain the following result.

Proposition 8. *Propositions 3–6 and Corollary 1 continue to hold if "faithful" is changed to "faithful wrt. a given set $E \supseteq env(\Pi)$" for a HEX program Π.*

Proof. The proof of Proposition 3 goes through for assignments $\mathbf{A}, \mathbf{A}'$ such that $\{a \mid \mathbf{T}a \in \mathbf{A} \cup \mathbf{A}'\} \subseteq env(\Pi)$ as well. Proposition 4 does not involve faithfulness and thus carries over. Next, the proofs of Corollary 1 and Proposition 5 also work for faithfulness wrt. Π, as the reference to Proposition 3 can be amended to its relativized version. Finally, the proof of Proposition 6 can be adapted by setting $\mathbf{A}_1(p_i(\vec{d})) = \mathbf{F}$ and $\mathbf{A}'_1(p_i(\vec{d})) = \mathbf{F}$ instead of $\mathbf{A}_1(p_i(\vec{d})) = \mathbf{T}$ and $\mathbf{A}'_1(p_i(\vec{d})) = \mathbf{T}$ in the "otherwise" case. □

Thus, in particular we can use the normal form and transform, relative to an "envelope" E for the set of atoms that are true in some answer set of the guessing program $\hat{\Pi}$, every set D of faithful io-dependencies into a tight faithful set D' of io-dependencies. Notably such a set D' may be strictly tighter than a tight faithful D'' in absence of an envelope (intuitively, there are fewer possibilities for a pair $\mathbf{A}, \mathbf{A}'$ of assignments refuting faithfulness). Thus, the dependency graph may be even pruned more, resulting in a further pruning of the search space of answer set candidates.

Example 17 *(Example 14, cont'd).* For the program Π in Example 3, we have that $env(\Pi) = \{a \mid \mathbf{T}a \in \mathbf{A}_0\} = \{city(vienna), location(bratislava), closeCity(bratislava), city(bratislava)\}$ as the assignment $\mathbf{A}_0$ is the single answer set of Π. Relative to assignments $\mathbf{A}$ such that $\{a \mid \mathbf{T}a \in \mathbf{A}\} \supseteq env(\Pi)$, the external atom &*closeTo*[*city*](*kobe*) always evaluates to false; thus the io-dependency $\langle\{osaka\}, kobe\rangle$, can be tightened to $\langle\emptyset, kobe\rangle$, similarly, the io-dependency $\langle\{kobe\}, osaka\rangle$, can be tightened

to $\langle \emptyset, osaka \rangle$, Relative to $env(\Pi)$, the set $D^\star = \{\langle \emptyset, kobe \rangle, \langle \emptyset, osaka \rangle, \langle \{vienna\}, bratislava \rangle, \langle \{bratislava\}, vienna \rangle\}$, is tightest faithful.

For the ground program Π' that is produced by the DLVHEX grounder for the program Π in Example 11, we know that $env(\Pi) \subseteq env(\Pi')$ holds, and clearly $env(\Pi') \subseteq \{node(a), node(b), node(c), node(d)\} =: E$ holds. Thus the io-dependency $\langle \{c, e\}, d \rangle$ in the specified set $D(\&suc[node])$ can be tightened to $\langle \{c\}, d \rangle$ as the input $node(e)$ is irrelevant for the value of $\&suc[node](d)$ relative to E (and thus relative to $env(\Pi)$).

The best (smallest) envelope according to Definition 11 is $E = env(\Pi)$, i.e., the set of all ground atoms that are bravely entailed by Π. However, to actually compute or use $env(\Pi)$ for faithfulness checks in order to optimize the evaluation of Π is pointless, because the original program must already be evaluated. Instead, one may use $E = env(\hat{\Pi})$, i.e., the atoms bravely entailed from the guessing program $\hat{\Pi}$ for Π; notice that $env(\Pi) \subseteq env(\hat{\Pi})$ holds since every answer set $\mathbf{A}$ of Π is contained in some compatible set $\hat{\mathbf{A}}$ for Π.

However, computing $env(\hat{\Pi})$ comes at a cost, and is under worst-case complexity not easier to compute than $env(\Pi)$, and thus intractable in general. This motivates us to find envelopes that can be computed in polynomial time, which we will address in Section 4.2.

4. Computational complexity

In this section, we consider computational complexity issues for testing faithfulness, where we first consider universal io-dependencies (which are agnostic about the program) and then relativized io-dependencies. We assume throughout tacitly that the (finite) set $\mathcal{C}$ of constants for the evaluation of programs is part of the input, and that evaluating ground external atoms, i.e., deciding whether $\mathbf{A} \models \&g[\vec{p}](\vec{c})$ holds, is feasible in polynomial time.

In the following, we call the arity of predicates $p \in \mathcal{P}$ *bounded*, if $ar(p) \leq k$ holds for some constant k.

4.1. Universal io-dependencies

To check faithfulness of a set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies, formally the oracle function $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c})$ must be evaluated for all evaluations of predicates $p \in \vec{p}$ and output tuples $\vec{c}$, which naively is often not feasible in practice.

Example 18. Reconsider $\&suc[node](X)$ from Example 10. To check faithfulness of the specified io-dependencies wrt. output a, the oracle function needs to be evaluated under all possible assignments to atoms with predicate $node$.

In the worst case, this cannot be avoided by the following result, where we assume that $\&g[\vec{p}](\vec{c})$ is decidable in polynomial time.

Theorem 2 *(Faithfulness complexity). Checking faithfulness of a given set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies is (i) co-*NEXPTIME*-complete in general, and (ii) co-*NP*-complete if each $p_i \in \vec{p}$ has bounded predicate arity. Furthermore, co-*NEXPTIME*-hardness holds in (i) even if D contains a single dependency and $\&g$ is unary (i.e., $ar_O(\&g) = 1$) with a single input predicate (i.e., $\vec{p} = p_1$), and co-*NP*-hardness holds in (ii) if in addition p_1 is unary.*

Proof. Membership in co-NEXPTIME respectively co-NP can be shown by a guess and check algorithm for disproving faithfulness of D: to this end, we can guess assignments $\mathbf{A}$, $\mathbf{A}'$ to the input predicates $\vec{p}$, and an output tuple $\vec{c}$, such that $\mathbf{A}$, $\mathbf{A}'$ coincide on all atoms in $comp(D, \&g[\vec{p}](\vec{c}))$ and $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) \neq f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.

The guess for $\mathbf{A}$, $\mathbf{A}'$ and $\vec{c}$ is in the general case of exponential size in the input, while it has polynomial size if the arity of the predicates p_i in $\vec{p}$ is bounded by a constant, as only a polynomial number of atoms in the size of the set of constants is possible.

In order to verify the guess, one first computes the set $comp(D, \&g[\vec{p}](\vec{c}))$; this is feasible in exponential (resp. polynomial) time in the size of the input. Checking whether $\mathbf{A}$, $\mathbf{A}'$ coincide on $comp(D, \&g[\vec{p}](\vec{c}))$ and computing $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c})$ and $f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$ is both feasible in exponential (resp. polynomial) time in the size of the input; overall, this means that disproving faithfulness of D is in NEXPTIME, from which the claimed upper bound follows.

For the hardness parts, we provide a reduction from the complement of *Graph 3-Colorability*, which is a canonical NP-complete problem. For succinct input representation, this problem is well-known to be NEXPTIME-complete [64]; that is, the graph $G = (V, E)$ is not given in the usual form (say, by the adjacency matrix of its vertices), but by a Boolean circuit C_G with $2n$ input bits $b_1, \ldots, b_{2n}$ such that $v = b_1, \ldots, b_n$ and $v' = b_{n+1}, \ldots, b_{2n}$ represent nodes (in binary coding) and the circuit C_G outputs 1 for v, v' if and only if there is an edge between v and v'. (Note that C_G can be exponentially more succinct than the usual representation of G, which intuitively explains the exponential complexity blowup.)

We reduce 3-colorability to non-faithfulness checking as follows. We use an external predicate $\&col_G[r, g, b]/1$ that has three n-ary input predicates $\vec{p} = r, g, b$, where each ground atom $r(b_1, \ldots, b_n)$ with all $b_i \in \{0, 1\}$ means that the vertex $v = b_1, \ldots, b_n$ is colored red (analogously for b and g). The function $f_{\&g}$ is defined as follows. For every assignment $\mathbf{A}$,

- $f_{\&col_G}(\mathbf{A}, \vec{p}, 0)$ evaluates to $\mathbf{T}$ iff $\vec{p} = r, g, b$ as given in $\mathbf{A}$ does not constitute a legal 3-coloring for the graph G;
- $f_{\&col_G}(\mathbf{A}, \vec{p}, 1)$ takes the opposite truth value.[3]

Notably, evaluating $f_{\&col_G}(\mathbf{A}, \vec{p}, 1)$ is feasible in polynomial time in the size of $\mathbf{A}$ and C_G: to this end, a Turing machine M_G may cycle through all pairs v, v' of nodes and simulate for each pair the evaluation of C_G and check whether v, v' are colored differently if an edge between them exists. Clearly, evaluating $f_{\&col_G}(\mathbf{A}, \vec{p}, 0)$ is then also feasible by M_G in polynomial time.

The machine M_G is constructible in polynomial time from C_G, and it runs on input $\mathbf{A}$ in time polynomial in the size of $\mathbf{A}$ and C_G. Thus, M_G constitutes the polynomial-time implementation of $f_{\&col_G}$ as required.

We now define io-dependencies $D = \{\delta_1, \delta_2, \delta_3\}$, where $\delta_i = \langle i, 1:\{0\}, 1:0\rangle$, for $i = 1, 2, 3$. Intuitively for output 0, only the color assignment to the vertices in $V_0 = \{v = 0, b_2, \ldots, b_n \mid b_i \in \{0, 1\},\ 2 \leq i \leq n\}$ matters, i.e., those with a leading 0 in the binary representation, as $comp(D, \&col_G[r, g, b](0)) = \{r(v), g(v), b(v) \mid v \in V_0\}$. However, to be sure that any 3-coloring for these vertices (which might be feasible) cannot be extended to a 3-coloring of all vertices, it must hold that the full graph is not 3-colorable.

Formally, we claim that D is faithful wrt. $\&col_G[r, g, b]$ iff the graph G is not 3-colorable.

($\Leftarrow$) Assume that G is not 3-colorable. Then, for every assignment $\mathbf{A}$, we have that $f_{\&col_G}(\mathbf{A}, r, g, b, 1) = \mathbf{F}$ and $f_{\&col_G}(\mathbf{A}, r, g, b, 0) = \mathbf{T}$; hence D is clearly faithful, as no counterexample to the faithfulness condition is possible.

($\Rightarrow$) Assume that G is 3-colorable. Then an assignment $\mathbf{A}$ exists s.t. $f_{\&col_G}(\mathbf{A}, r, g, b, 1) = \mathbf{T}$ holds, which means $f_{\&col_G}(\mathbf{A}, r, g, b, 0) = \mathbf{F}$. However, for the assignment $\mathbf{A}'$ that coincides with $\mathbf{A}$ on V_0 and assigns no color to the remaining vertices $V \setminus V_0$ (where without loss of generality, some such vertex exists), we have $f_{\&col_G}(\mathbf{A}, r, g, b, 1) = \mathbf{F}$ and thus $f_{\&col_G}(\mathbf{A}, r, g, b, 0) = \mathbf{T}$; hence, D is not faithful.

This shows the co-NExpTime-hardness of the problem in case (i). In the case (ii) where the predicate arities are bounded by a constant, we use the nodes V as constants, and $r(v)$ expresses that vertex v is colored red (analogously for b and g); without loss of generality, we have that $V = \{0, \ldots, |V|-1\}$. Then, assuming that only 0 and 1 can be in the output of $\&col_G[r, g, b]$, i.e., $f_{\&col_G}(\mathbf{A}, r, g, b, c) = \mathbf{F}$ for every $c \neq 0, 1$, we similarly conclude that D is not faithful wrt. $\&col_G[r, g, b]$ iff G is 3-colorable; this proves co-NP-hardness for case (ii).

It remains to establish the hardness parts under the stated restrictions. To this end, we change the representation of the graph coloring and merge r, g, b into a single n+2-ary predicate p, by encoding the color in the last two bits of a tuple $b_1, \ldots, b_n, b_{n+1}, b_{n+2}$ such that $b_{n+1}, b_{n+2} = 0, 0$ represents r, $b_{n+1}, b_{n+2} = 0, 1$ represents g, and $b_{n+1}, b_{n+2} = 1, 0$ represents b; the code $b_{n+1}, b_{n+2} = 1, 1$ is unused, i.e., any such tuple can be ignored. The Turing machine M_G for checking graph 3-(un)colorability from above is easily adjusted to this encoding. It then holds that $D' = \{\langle 1, 1:\{0\}, 1:0\rangle\} \subseteq dep(\&col'_G[p])$, where col'_G is the adjusted external predicate, is faithful iff G is not 3-colorable. This establishes co-NExpTime-hardness under the restriction for the case (i). For the bounded predicate case, we similarly merge r, g, p into a unary predicate p over domain $V \times \{r, g, b\}$ that is encoded by the integers $\{0, \ldots, |V|^3 - 1\}$; then the same D' is faithful iff G is not 3-colorable. This proves the claim. □

When certain properties of external sources are known, less external calls are needed for faithfulness checking, e.g. for monotonic functions. An input $p_i \in \vec{p}$ of a ge-predicate $\&g[\vec{p}]$ is *monotonic*, if for any assignment $\mathbf{A}$ and output $\vec{c}$, $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = \mathbf{T}$ implies $f_{\&g}(\mathbf{A}', \vec{p}, \vec{c}) = \mathbf{T}$ for every $\mathbf{A}' \geq \mathbf{A}$ such that $\mathbf{A}(p_j(\vec{d})) = \mathbf{A}'(p_j(\vec{d}))$ for all predicates $p_j \in \vec{p}$ with $p_j \neq p_i$ (cf. [28]). Based on monotonicity, the number of assignments to consider in a faithfulness check can be decreased.

Proposition 9 *(Monotonic Faithfulness). If $p_i \in \vec{p}$ for $\&g[\vec{p}]$ is monotonic, a set $D \subseteq dep(\&g[\vec{p}])$ is* faithful for $\&g[\vec{p}]$ *iff for any assignments $\mathbf{A}$, $\mathbf{A}'$ such that (i) $\mathbf{T}p_i(\vec{d}) \in \mathbf{A}$ and $\mathbf{F}p_i(\vec{d}) \in \mathbf{A}'$ for every $p_i(\vec{d}) \notin comp(D, \&g[\vec{p}](\vec{c}))$ and (ii) $\mathbf{A}(p_i(\vec{d})) = \mathbf{A}'(p_i(\vec{d}))$ for every $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$, it holds that $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.*

Proof. Let p_i be a monotonic input parameter of a ge-predicate $\&g[\vec{p}]$, and $D \subseteq dep(\&g[\vec{p}])$. We need to show that D is faithful for $\&g[\vec{p}]$ iff for any two assignments $\mathbf{A}$, $\mathbf{A}'$ such that $\mathbf{T}p_i(\vec{d}) \in \mathbf{A}$ and $\mathbf{F}p_i(\vec{d}) \in \mathbf{A}'$ for every atom $p_i(\vec{d}) \notin comp(D, \vec{c})$, and $\mathbf{A}(p_i(\vec{d})) = \mathbf{A}'(p_i(\vec{d}))$ for every atom $p_i(\vec{d}) \in comp(D, \vec{c})$, it holds that $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.

($\Rightarrow$) The only-if direction follows directly from Definition 6.

($\Leftarrow$) To prove the if-direction, (*) assume that for any two assignments $\mathbf{A}$, $\mathbf{A}'$ s.t. $\mathbf{T}p_i(\vec{d}) \in \mathbf{A}$ and $\mathbf{F}p_i(\vec{d}) \in \mathbf{A}'$ for every atom $p_i(\vec{d}) \notin comp(D, \vec{c})$, and $\mathbf{A}(p_i(\vec{d})) = \mathbf{A}'(p_i(\vec{d}))$ for every atom $p_i(\vec{d}) \in comp(D, \vec{c})$, it holds that $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$. We need to show that D is faithful for $\&g[\vec{p}]$ according to Definition 6, i.e. that for any two assignments $\mathbf{A}_*$, $\mathbf{A}'_*$ and any possible output tuple $\vec{c}$ for $\&g[\vec{p}]$ it holds that if $\mathbf{A}_*(p_i(\vec{d})) = \mathbf{A}'_*(p_i(\vec{d}))$ for every atom $p_i(\vec{d}) \in comp(D, \vec{c})$, then $f_{\&g}(\mathbf{A}_*, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}'_*, \vec{p}, \vec{c})$.

First consider the case that $f_{\&g}(\mathbf{A}_*, \vec{p}, \vec{c}) = \mathbf{T}$. Let $\mathbf{A}$ be an assignment s.t. $\mathbf{A}_*(p_j(\vec{d})) = \mathbf{A}(p_j(\vec{d}))$ for all atoms $p_j(\vec{d})$ with $p_j \in \vec{p}$ where $p_j \neq p_i$ and all atoms $p_j(\vec{d}) \in comp(D, \vec{c})$ where $p_j = p_i$, and $\mathbf{T}p_i(\vec{d}) \in \mathbf{A}$ for every atom $p_i(\vec{d}) \notin comp(D, \vec{c})$. Furthermore, let $\mathbf{A}'$ be an assignment s.t. $\mathbf{A}'_*(p_j(\vec{d})) = \mathbf{A}'(p_j(\vec{d}))$ for all atoms $p_j(\vec{d})$ with $p_j \in \vec{p}$ where $p_j \neq p_i$ and all atoms

[3] For the proof, it would be sufficient to let $f_{\&col_G}(\mathbf{A}, \vec{p}, 1)$ take an arbitrary but fixed truth value; however, we prefer here to model $f_{\&g}$ as a total single-valued function.

$p_j(\vec{d}) \in comp(D, \vec{c})$ where $p_j = p_i$, and $\mathbf{F}p_i(\vec{d}) \in \mathbf{A}'$ for every atom $p_i(\vec{d}) \notin comp(D, \vec{c})$. Since $f_{\&g}(\mathbf{A}_*, \vec{p}, \vec{c}) = \mathbf{T}$, we obtain $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = \mathbf{T}$ according to the definition of monotonic input parameter. Moreover, it follows from our assumption (*) that $f_{\&g}(\mathbf{A}', \vec{p}, \vec{c}) = \mathbf{T}$. Finally, $f_{\&g}(\mathbf{A}'_*, \vec{p}, \vec{c}) = \mathbf{T}$ follows again from the definition of monotonic input parameter because we have that $f_{\&g}(\mathbf{A}', \vec{p}, \vec{c}) = \mathbf{T}$, which proves that $f_{\&g}(\mathbf{A}_*, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}'_*, \vec{p}, \vec{c})$.

The case for $f_{\&g}(\mathbf{A}_*, \vec{p}, \vec{c}) = \mathbf{F}$ works analogously. □

Example 19 *(Example 18 cont'd).* As the predicate *node* is a monotonic input parameter of &*suc*[*node*], for checking faithfulness wrt. a it suffices to evaluate $f_{\&suc}(\mathbf{A}, node, a)$ under two assignments $\mathbf{A}_t$ and $\mathbf{A}_f$, such that

- $\mathbf{A}_t \subseteq \{\mathbf{T}node(a), \mathbf{T}node(b), \mathbf{T}node(c), \mathbf{T}node(d)\}$ and
- $\mathbf{A}_f \subseteq \{\mathbf{F}node(a), \mathbf{F}node(b), \mathbf{F}node(c), \mathbf{F}node(d)\}$.

Under additional conditions, we obtain tractability:

Corollary 3. *Suppose all $p_i \in \vec{p}$ for $\&g[\vec{p}]$ are monotonic and have bounded arities. Then checking faithfulness of $D \subseteq dep(\&g[\vec{p}])$ for $\&g[\vec{p}](\vec{c})$ is polynomial if $|comp(D, \&g[\vec{p}](\vec{c}))|$ is bounded by a constant.*

Proof. Indeed, if we have that $|comp(D, \&g[\vec{p}](\vec{c})| \leq k$ for a constant k, then after computing $comp(D, \&g[\vec{p}](\vec{c}))$, which can for bounded predicate arities be done in polynomial time by cycling through all (polynomially many) atoms, we need to consider by Proposition 9 for each assignment on $comp(D, \&g[\vec{p}](\vec{c}))$ only the two specific assignments that set all other ground atoms to true resp. all to false; thus, we need to evaluate and compare $2{\cdot}2^k = 2^{k+1}$ function calls, which is feasible in polynomial time. □

The same holds for computing a tight faithful set D for $\&g[\vec{p}](\vec{c})$. In practice, this applies to Example 10, if the external graph has bounded degree.

4.1.1. Faithful guessing program

As we have remarked in Section 3, the concept of faithful io-dependencies is a sufficient but not a necessary criterion for omitting the external minimality check when the answer sets of a program Π are computed from those of the guessing program $\hat{\Pi}$, i.e., the property (FGP). Testing this property has the following complexity.

Theorem 3. *Deciding whether a given* HEX *program Π fulfills property (FGP) is (i) co-*NEXPTIME$^{\text{NP}}$*-complete in general and (ii) Π_3^p-complete if all predicates occurring in Π have bounded predicate arities.*

Proof. As for the membership parts, to refute property (FGP) we can guess $\hat{\mathbf{A}}$ such that (a) $\hat{\mathbf{A}}$ is an answer set of the guessing program $\hat{\Pi}$ and (b) $\mathbf{A} = \hat{\mathbf{A}}|_\Pi$ is not an answer set of Π. The size of $\hat{\mathbf{A}}$ and $\mathbf{A}$ is exponential in the size of Π in general. Furthermore, checking (a) is in co-NP given $\hat{\mathbf{A}}$ and $grnd(\hat{\Pi})$, since (a.1) deciding whether $\hat{\mathbf{A}}$ violates $f\hat{\Pi}^{\hat{\mathbf{A}}}$ is polynomial (find some rule $r \in grnd(\hat{\Pi})$ such that $\hat{\mathbf{A}} \models B(r)$ and $\hat{\mathbf{A}} \not\models H(r)$) and (a.2) deciding whether some $\hat{\mathbf{A}}' \subset \hat{\mathbf{A}}$ satisfies $f\hat{\Pi}^{\hat{\mathbf{A}}}$ is in NP. Along similar lines, we can conclude that checking (b) is in NP given $\mathbf{A}$ and $grnd(\Pi)$ as external atoms can be evaluated in polynomial time. As $grnd(\hat{\Pi})$ and $grnd(\Pi)$ can be constructed in exponential time from Π, it follows that (FGP) can be refuted in nondeterministic exponential time with an NP oracle; this proves the membership part for (i).

As for (ii), the guess for $\hat{\mathbf{A}}$ has polynomial size if predicate arities are bounded, and deciding (a) is in Π_2^p [21] (as (a.1) is in co-NP and (a.2) is in Σ_2^p) and (b) is in Σ_2^p along similar lines. Thus, (FGP) can be refuted in nondeterministic polynomial time with an Σ_2^p oracle, which proves the membership part of (ii).

To establish the hardness parts, it is sufficient to add to an ordinary disjunctive logic program Π a rule $p(a) \leftarrow \&id[p](a)$ as in Example 5, where p is a fresh predicate. Then the guessing program $\hat{\Pi}'$ for the resulting program Π' satisfies (FGP) iff Π has no answer set. The latter is co-NEXPTIME$^{\text{NP}}$-complete in general and Π_3^p-complete in case of bounded predicate arities [21]. □

Thus, testing (FGP) is more expensive than testing a set of io-dependencies in the worst case and thus ill-suited for the aim of improving performance. We note that also for disjunction-free programs, testing (FGP) is by the reduction in the proof of Theorem 3 intractable (at least Π_2^p-hard in the bounded predicate case). On the other hand, the external atom $\&id[p](a)$ occurring in the reduction is monotonic and p has bounded arity. Thus, for a small set D of io-dependencies, in particular for $D = \{\langle 1, 1 : \{a\}, a\rangle\}$, testing faithfulness is polynomial according to Corollary 3.

4.2. Relativized io-dependencies

Checking faithfulness of a set D of io-dependencies wrt. an envelope E does not get more complex in general, if E is part of the input, since E can be used to constrain guesses for assignments that witness the failure of faithfulness. If an envelope E can be constructed efficiently, then using E adds only polynomial overhead in the worst case; furthermore, having an envelope may under certain conditions make faithfulness testing even easier.

Proposition 10. *Deciding whether a given set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies is faithful with respect to a given envelope E is co-*NP*-complete, if deciding $\mathbf{A} \models \&g[\vec{p}](\vec{c})$ is for every $\vec{c}$ feasible in polynomial time in the cardinality of $\{a \mid \mathbf{T}a \in \mathbf{A}\}$.*

Proof. Indeed, guesses for assignments $\mathbf{A}$, $\mathbf{A}'$ that witness the failure of faithfulness of D can be constrained with E and represented by their true-parts $\{a \mid \mathbf{T}a \in \mathbf{A}\}, \{a \mid \mathbf{T}a \in \mathbf{A}'\} \subseteq E$, which have polynomial size; furthermore, $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c})$ and $f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$ can be then computed and compared in polynomial time. Thus, the problem is in co-NP. The co-NP-hardness is inherited from the bounded predicate case (item (ii) of Theorem 2), since E can be set to the Herbrand base of $\vec{p}$ and $\mathcal{C}$, which is computable in polynomial time in this case. □

Note that this result holds for arbitrary (unbounded) predicate arity; for example, deciding faithfulness of the dependencies $D \subseteq dep(\&col_G[r, g, b])$ in the proof of Theorem 2, which is co-NExpTime-complete, drops to co-NP if this is done relative to an envelope E that contains colors of the nodes. However, E may be exponential in the representation of D and $f_{\&\&col_G[r,g,b]}$, and thus the complexity drop is only artificial.

4.2.1. Guessing envelope

In Section 3.3, we have introduced $env(\hat{\Pi})$ as a particular envelope. Computing the latter is under worst-case complexity not easier than computing $env(\Pi)$, and is thus intractable in general. It subsumes for programs with polynomial-time external atoms ($co-$)NExpTime$^{\text{NP}}$-complete problems in general and Σ_3^p/Π_3^p-complete problems for bounded predicate arities.

On the other hand, $env(\hat{\Pi})$ need not be provided in the input but relevant bits of it may be computed on the fly. Nonetheless, the complexity of cautious reasoning from an ordinary logic program still dominates the complexity of faithfulness checking.

Theorem 4. *Checking faithfulness of a given set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies relative to $env(\hat{\Pi})$ for a given* HEX *program Π is co-*NExpTime$^{\text{NP}}$*-complete in general, and Π_3^p-complete for fixed predicate arities.*

Proof. Membership in co-NExpTime$^{\text{NP}}$ respectively Π_2^p can be shown by a guess and check algorithm for disproving faithfulness of D: to this end, we can as in the proof of Theorem 2 guess assignments $\mathbf{A}$, $\mathbf{A}'$ to the input predicates $\vec{p}$, and an output tuple $\vec{c}$, such that $\mathbf{A}$, $\mathbf{A}'$ coincide on all atoms in $comp(D, \&g[\vec{p}](\vec{c}))$ and $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) \neq f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.

Recall that the guess for $\mathbf{A}$, $\mathbf{A}'$ and $\vec{c}$ is in the general case of exponential size in the input, while it has polynomial size if the arity of the predicates p_i in $\vec{p}$ is bounded by a constant. To check in addition to the tests in the proof of Theorem 2 whether each atom $p(\vec{c})$ that is true in $\mathbf{A}$ resp. $\mathbf{A}'$ is in $env(\hat{\Pi})$, we can guess for $p(\vec{c})$ an answer set $\mathbf{A}$ of $\hat{\Pi}$ that contains $p(\vec{c})$. Verifying the latter guess is possible using an NP oracle in the unrestricted case, and using a Σ_2^p oracle in the bounded predicate arity case; summarizing, this shows membership of disproving faithfulness in NExpTime$^{\text{NP}}$ for arbitrary programs and in Σ_3^p for programs with bounded predicate arities, from which the claimed membership results follow.

For the hardness parts, we provide a reduction from the complement of brave reasoning from an ordinary logic program Π. That is, given such a program Π and a ground atom A, decide whether no answer set $\mathbf{A}$ of Π exists such that $\mathbf{T}A \in \mathbf{A}$ holds. This problem is co-NExpTime$^{\text{NP}}$-complete for unrestricted programs [26] and Π_3^p-complete for programs under bounded predicate arities [21]. Without loss of generality, we may assume that A is of the form $p(a)$.

We define an external atom $\&e[p](x)$, where p is unary, such that $f_{\&e}(\mathbf{A}, p, c) = 1$ iff $p(c) \in \mathbf{A}$ holds, and define $D = \{\langle 1, 1 : \emptyset, 1 : a\rangle\}$; informally, the dependency states that a is either always or never in the output of $\&e[p]$, regardless of the interpretation $\mathbf{A}$; note that $comp(D, \&e[p](a)) = \emptyset$. It is then easy to check that

D is faithful relative to Π
  iff no assignments $\mathbf{A}, \mathbf{A}'$ exists s.t. $\{a, a' \mid \mathbf{T}a \in \mathbf{A}, \mathbf{T}a' \in \mathbf{A}'\} \subseteq env(\hat{\Pi})$
    and $f_{\&e}(\mathbf{A}, p, a) \neq f_{\&e}(\mathbf{A}', p, a)$
  iff $env(\hat{\Pi})$ does not contain $p(a)$
  iff $p(a)$ is not a brave consequence of $\hat{\Pi}$
  iff $p(a)$ is not a brave consequence of Π.

This proves co-NExpTime$^{\text{NP}}$-hardness for arbitrary programs and Π_3^p-hardness for programs with bounded predicate arities, respectively. □

We observe that the complexity drops to co-NExpTime resp. Π_2^p if the program Π is disjunction-free (i.e., normal); in this case $\hat{\Pi}$ is readily rewritten to a normal program, by replacing the guessing clauses $e_{\&g[\vec{p}]}(\vec{t}) \vee ne_{\&g[\vec{p}]}(\vec{t}) \leftarrow$ with rules[4]

$$e_{\&g[\vec{p}]}(\vec{t}) \leftarrow \text{not}\, ne_{\&g[\vec{p}]}(\vec{t}). \qquad ne_{\&g[\vec{p}]}(\vec{t}) \leftarrow \text{not}\, e_{\&g[\vec{p}]}(\vec{t}).$$

Cautious reasoning from this program is then in co-NExpTime resp. Π_2^p, and also complete for this class. This shows that for fragments of HEX programs with polynomial-time evaluable external atoms, testing faithfulness of a set D of io-dependencies

[4] Safety of non-ground rules can be ensured as usual by adding for the variables X occurring in the head domain predicates $dom(X)$ in the body.

wrt. $env(\hat{\Pi})$ may have lower complexity than cautious reasoning from the program Π (under common complexity hypotheses). In particular, this holds for Horn-style programs (i.e., without disjunction and negation) with bounded predicate arities, for which answer set existence is still Σ_3^p-complete. This can be seen because negation not $p(\vec{X})$ and disjunction $p(\vec{X}) \vee q(\vec{X})$ in rule heads (which allow constructing worst-case complexity instances) can be expressed using polynomial-time evaluable external atoms. Indeed,

- we can express not $p(\vec{X})$ with an atom $\&naf[p](\vec{X})$, where $f_{\&naf}(\mathbf{A}, p, c) = \mathbf{T}$ iff $\mathbf{T}p(c) \notin \mathbf{A}$, and eliminate negative literals from Π;
- furthermore, we can use for a disjunction $p(\vec{X}) \vee q(\vec{X})$ in the head of a rule $r : H \leftarrow B$ an external atom $\&dis_l[p, q](X)$, such that $f_{\&dis_l}(\mathbf{A}, p, q, c) = \mathbf{T}$ iff $\mathbf{T}p(c) \in \mathbf{A}$ or $\mathbf{F}q(c) \in \mathbf{A}$, and replace r with two rules $r_1 : p(X) \leftarrow \&dis_l[p, q](X), B$ and $r_2 : q(X) \leftarrow \&dis_l[q, p](X), B$.[5]

If a HEX program Π where all disjunctions are of this form is rewritten to a Horn-style HEX program Π', then the answer sets of Π' correspond to the answer sets of Π (for more details, see Appendix).

4.2.2. Head and Horn envelope

In practice, one thus looks for envelopes that can be computed more efficiently. Clearly, in case the program Π is ground, we may simply take the head envelope $E_{hd}(\Pi) = \bigcup_{r \in \Pi} H(r)$; while this does not decrease the worst-case complexity the reduction of the search space can be significant.

A more restrictive envelope is the *Horn envelope* of Π, which is defined as follows.

Definition 12 *(Horn Envelope).* Given a HEX program Π, the *Horn envelope* of Π, denoted $env_H(\Pi)$, is defined as $env_H(\Pi) = env(\Pi_H)$, where Π_H is obtained from Π

1. by dropping all constraints,
2. by splitting up all disjunctive rule heads, and
3. by removing all negative literals and all external atoms.[4]

Note that $env(\Pi_H) = \{a \mid \mathbf{T}a \in \mathbf{A}\}$ where $\mathbf{A}$ is the unique answer set of Π_H, which coincides with the least model of Π_H.

Example 20. Reconsider the program Π in Example 3; then Π_H consists of the rules

$$\begin{aligned} city(X) &\leftarrow closeCity(X), \\ closeCity(X) &\leftarrow location(X), \\ city(vienna), & \\ location(bratislava) & \end{aligned}$$

and thus we have that $env_H(\Pi) = \{city(vienna), location(bratislava), closeCity(bratislava), city(bratislava)\}$. Note that the head envelope of Π is $env_{hd}(\Pi) = env_H(\Pi) \cup \{city(kobe), city(osaka), closeCity(kobe), closeCity(osaka), closeCity(vienna)\}$.

Notably the Horn envelope of the original program and the guessing program are equally good for checking faithfulness.

Proposition 11. *For every* HEX *program* Π, *it holds that* $E_H(\Pi) = E_H(\hat{\Pi})|_\Pi$.

Proof. Recall that $\hat{\Pi}$ results from Π by replacing every atom $\&g[\vec{p}](\vec{c})$ in it by an ordinary *replacement atom* $e_{\&g[\vec{p}]}(\vec{c})$, and by adding a guessing rule $e_{\&g[\vec{p}]}(\vec{c}) \vee ne_{\&g[\vec{p}]}(\vec{c}) \leftarrow$. In the program $\hat{\Pi}_H$, the latter is replaced by the two facts $e_{\&g[\vec{p}]}(\vec{c})$, and $ne_{\&g[\vec{p}]}(\vec{c})$, which then mimics the removal of a positive occurrence $\&g[\vec{p}](\vec{c})$. A negative occurrence not $\&g[\vec{p}](\vec{c})$, which is removed from Π, leads to not $e_{\&g[\vec{p}]}(\vec{c})$, which is removed from $\hat{\Pi}$ alike. The claim easily follows. □

For ground programs Π, clearly $E_H \subseteq E_{hd}$ holds; furthermore E_H is computable in polynomial time by evaluation of the ground program Π_H. For non-ground programs, the Horn envelope $E_H(\Pi)$ is computable in exponential time in general (by grounding and solving), while in case of bounded predicate arities, $E_H(\Pi)$ can be computed by fixpoint iteration in polynomial time with an NP oracle (in fact, even with parallel oracle calls). However, not the full Horn envelope is needed for deciding faithfulness.

[5] Intuitively, if both $p(X)$ and $q(X)$ are false in $\mathbf{A}$, some atom in B must be false in $\mathbf{A}$ and thus r_1, r_2 are dropped in the FLP-reduct wrt. $\mathbf{A}$. Otherwise, if B is true in $\mathbf{A}$, some rule r_i is in the FLP-reduct, and a smaller model $\mathbf{A}' < \mathbf{A}$ of the FLP-reduct in which $p(X)$, $q(X)$ are false can only exist if also B is false in $\mathbf{A}'$.

Proposition 12. *Deciding whether a given set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies is faithful with respect to the Horn envelope of a given program Π is (i) co-*NEXPTIME*-complete in general and (ii) co-*NP*-complete if all predicate arities in Π are bounded.*

Proof. The faithfulness of D can be rejected with assignments $\mathbf{A}$, $\mathbf{A}'$ such that for some $E' \subseteq E_H(\Pi)$ and output tuple $\vec{c}$ of $\&g[\vec{p}]$ it holds that (a) $\{a \mid \mathbf{T}a \in \mathbf{A} \cup \mathbf{A}'\} \subseteq E'$, (b) $\mathbf{A}(p_i(\vec{d})) = \mathbf{A}'(p_i(\vec{d}))$ for every atom $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ and (c) $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) \neq f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$. A guess for E' has size at most exponential in the input size, and each atom $a \in E'$ can be witnessed by a derivation from Π_H, which is constructible in polynomial time from Π, that can be guessed and verified in exponential time. Furthermore, $\vec{c}$ and $\mathbf{A}$, $\mathbf{A}'$ as in (a) can be guessed in exponential time, and conditions (b) and (c) can be verified in exponential time. This proves membership in co-NEXPTIME. The co-NEXPTIME-hardness is obtained from Theorem 2, as a program Π such that $E_H(\Pi)$ contains all ground atoms is easily defined. In the bounded arity case, all guesses have polynomial size and the refutation is feasible in polynomial time; the co-NP-hardness follows again from Theorem 2. □

If in addition to bounded predicate arities the rule bodies in Π are short (i.e., each rule r has $|B(r)|$ bounded by a constant) or amount to acyclic conjunctive queries (or more generally, have bounded tree-width [16]), then computing $E_H(\Pi)$ is feasible in polynomial time; the reason is that while matching an ordinary rule body against a set of facts is well-known to be NP-complete in general [15], the problem is feasible in polynomial time for such instances. On the other hand, checking faithfulness relative to the Horn envelope $E_H(\Pi)$ remains co-NP-hard, as can be seen from the proof of Proposition 12. Under the further restrictions of Corollary 3, we however obtain tractability.

Proposition 13. *Suppose that in a* HEX *program Π all predicates have bounded arities, that rule bodies amount to queries with bounded tree-width [16], and that the external predicate $\&g[\vec{p}]$ is monotonic. Then checking faithfulness of $D \subseteq dep(\&g[\vec{p}])$ for $\&g[\vec{p}](\vec{c})$ is polynomial if $|comp(D, \&g[\vec{p}](\vec{c}))|$ is bounded by a constant.*

Proof. Under the given restrictions on Π, we can compute Π_H and then $E_H(\Pi)$ as the single answer set of Π_H by standard fixpoint iteration in polynomial time, since matching the body $B(r)$ of a rule in Π_H against an assignment $\mathbf{A}$ is feasible in polynomial time [16]. Similar as in the proof of Corollary 3, we can then compute $|comp(D, \&g[\vec{p}](\vec{c}))|$ and check, using Proposition 9, faithfulness in polynomial time. □

5. Implementation and evaluation

In this section, we briefly describe an implementation of the new technique for pruning e-minimality checks and we report on an experimental evaluation on a suite of benchmark problems.

5.1. Integration into DLVHEX

We have integrated the new pruning technique into the HEX-solver DLVHEX 2.5.0, which uses GRINGO 4.4.0 and CLASP 3.1.1 as backends [41].

Io-dependencies for external atoms are specified by plugin-methods that compute whether a dependency wrt. given input and output values exists. For each external atom, we can implement a *Python*-method

```
checkIoDependencyN(path,i,j,k,inp,outp),
```

where `N` is an identifier number for the particular check. The method is declared as the dependency check for an external atom in the respective plugin-implementation by providing the identifier `N` as a property.

The arguments `i`, `j` and `k` of `checkIoDependencyN(path,i,j,k,inp,outp)` correspond to the argument positions of an io-dependency $\langle i, j{:}J, k{:}e\rangle$, and `inp` and `outp` represent input and output constants. The argument `path` is used to provide additional information that may be required for computing dependencies, e.g. the path to a file containing an external graph used by the plugin. Given a ground external atom $\&g[\vec{p}](\vec{c})$, the *Python*-method checks whether `inp` $\in J$ for all $\langle i, j{:}J, k{:}\texttt{outp}\rangle \in D(\&g[\vec{p}])$. During the construction of the atom dependency graph as in Definition 8 by DLVHEX, the method `checkIoDependencyN` is called repeatedly to ensure that only compliant atoms are considered when computing external dependencies.

To activate dependency pruning based on specified io-dependencies, the DLVHEX-solver is executed with the command line option `-useatomcompliance`. The solver implementation incorporating our new method is online available.[6]

5.2. Experiments

For our experiments, we used a Linux machine with two 12-core AMD Opteron 6238 SE CPUs and 512 GB RAM; the timeout was 300 secs and the memout 8 GB per instance. The average runtime of 10 instances per problem size is reported

[6] https://github.com/hexhex/core/.

$$y\_nd(X) \vee n\_nd(X) \leftarrow domain(X). \qquad \leftarrow nd(X), nd\_f(X).$$
$$nd(X) \leftarrow y\_nd(X). \qquad \leftarrow \text{not } nd(X), nd\_a(X).$$
$$nd(X) \leftarrow \&hasAccess[nd](X). \qquad \leftarrow \#count\{X : y\_nd(X)\} > 3.$$

Fig. 2. User Access Selection rules.

(in secs) for computing all answer sets; the numbers in parentheses indicate for how many of the 10 instances per size the timeout was reached. In the result tables, the best running time for each problem size is highlighted in bold. All instances and benchmark results are online available.[7]

5.2.1. Experimental setup

Next, we describe the different benchmark settings as well as the problems used in our experiments and corresponding hypotheses. Our experiments also consider interactions of dependency graph pruning with previously developed techniques for evaluating external sources early during the search for compatible sets instead of only wrt. candidate answer sets [28].

Configurations. To gain insights into how dependency graph pruning and other techniques interact, we consider the frequency of external calls as a further factor. While basic evaluation in Section 2 evaluates external atoms wrt. candidate models, we can evaluate them also wrt. partial assignments. To make this possible, Eiter et al. [28] extended the classical notion of Boolean assignments to three-valued assignments. Based on the latter, external sources can be evaluated at any point during search and their truth values can be propagated. Eiter et al. [28] showed that this can significantly speed the search for compatible sets as well as the search for unfounded sets during the e-minimality check. Moreover, after each external evaluation, the outputs for the respective (partial) inputs are learned by the solver to avoid making the same wrong guess for the evaluation of an external atom multiple times.

In this regard, investigating how e-minimality check skipping interacts with partial evaluation is of interest because early external evaluation can speed up model search as well as the e-minimality check and thus, can potentially influence the impact of our new technique.

We compared three different configurations, each with and without dependency graph pruning based on specified io-dependencies (where +**io-dep** signifies that dependency graph pruning was activated):

- **c-mod**: external atoms are only evaluated wrt. *candidate models* (representing the standard configuration of DLVHEX);
- **part**: external atoms are evaluated wrt. *partial assignments* after every solver guess during the model search and the results are propagated; and
- **min-part**: external atoms are evaluated wrt. *partial assignments* after every solver guess during the *e-minimality check* and the results are propagated.

In the result tables, we show combinations of configurations where interactions are expected.

Benchmark problems. We utilized three different problems for our experiments on e-minimality check skipping, where the absence of cyclic dependencies involving external atoms can be exploited when io-dependencies are specified:

- A *User Access Selection* problem concerning the assignment of access rights wrt. nodes of a computer network, where constraints are imposed based on reachability of nodes within the network.
- *Sequential Allocation of Indivisible Goods* as considered by Kalinowski et al. [53], where agent preferences over a set of goods are interfaced via an external atom.
- Computing *Strategic Companies*, a well-known Σ_2^p-hard problem from the business domain introduced by Cadoli et al. [10], where we utilize an external atom to access externally stored ownership relations between companies.

Hypotheses. Regarding the effect of dependency graph pruning based on semantic dependencies, we made the following predictions:

(H1) Adding +**io-dep** to any configuration decreases the running time if e-cycles can be removed from the dependency graph.
(H2) The speedup is larger when +**io-dep** is combined with **part** and smaller when combined with **min-part**, whenever partial evaluation is beneficial.
(H3) If pruning does not skip e-minimality checks, no significant overhead in terms of running time is incurred when +**io-dep** is added.

[7] See www.kr.tuwien.ac.at/research/projects/inthex/dep-pruning.

Table 1
Results for User Access Selection (all answer sets; few cycles).

#	All Answer Sets						#cyclic
	c-mod	**c-mod +io-dep**	**part**	**part +io-dep**	**min-part**	**min-part +io-dep**	
10	0.46 (0)	**0.43** (0)	0.60 (0)	0.58 (0)	1.53 (0)	1.36 (0)	7/10
15	2.64 (0)	**2.18** (0)	4.58 (0)	3.91 (0)	7.41 (0)	4.43 (0)	3/10
20	16.43 (0)	**14.71** (0)	44.90 (0)	41.93 (0)	43.87 (0)	31.03 (0)	5/10
25	43.85 (0)	**38.25** (0)	102.39 (1)	93.65 (1)	81.51 (0)	67.59 (0)	5/10
30	110.24 (2)	**91.01** (2)	192.48 (4)	180.58 (4)	168.80 (2)	99.53 (2)	4/10
35	111.62 (1)	**79.69** (1)	217.58 (4)	178.62 (2)	161.86 (2)	83.18 (1)	3/10
40	189.64 (2)	**141.12** (2)	262.35 (6)	231.22 (5)	202.95 (3)	143.12 (2)	5/10
45	264.04 (5)	216.89 (4)	269.49 (6)	227.88 (5)	263.40 (5)	**202.55** (4)	5/10
50	300.00 (10)	227.15 (4)	300.00 (10)	249.55 (6)	300.00 (10)	**220.61** (3)	2/10

Table 2
Results for User Access Selection (all answer sets; many cycles).

#	All Answer Sets						#cyclic
	c-mod	**c-mod +io-dep**	**part**	**part +io-dep**	**min-part**	**min-part +io-dep**	
10	0.41 (0)	0.41 (0)	**0.35** (0)	0.36 (0)	0.46 (0)	0.46 (0)	10/10
15	7.55 (0)	7.61 (0)	**6.17** (0)	6.38 (0)	7.95 (0)	8.15 (0)	10/10
20	44.03 (1)	43.92 (1)	**6.52** (0)	6.57 (0)	44.54 (1)	44.66 (1)	10/10
25	107.50 (2)	107.95 (2)	**51.60** (1)	51.62 (1)	87.53 (1)	87.51 (1)	10/10
30	84.97 (0)	84.64 (0)	**44.23** (0)	44.73 (0)	85.64 (0)	85.42 (0)	10/10
35	223.56 (5)	222.95 (5)	**111.29** (1)	110.98 (1)	223.26 (5)	224.26 (5)	10/10
40	268.27 (7)	268.73 (7)	**152.53** (1)	153.28 (1)	268.86 (7)	269.44 (7)	10/10
45	284.12 (8)	284.33 (8)	**251.08** (4)	252.54 (4)	286.90 (8)	286.56 (8)	10/10
50	300.00 (10)	300.00 (10)	300.00 (10)	**298.61** (9)	300.00 (10)	300.00 (10)	10/10

5.2.2. Details and results

For the benchmark problems, which we will now describe in more detail, we obtained several results that are summarized in Tables 1–5.

User Access Selection (UAS). Consider a computer network (C, A) represented by a set of computer nodes C and a set of directed access connections A between nodes, where $n_1 \to n_2 \in A$, for $n_1, n_2 \in C$, holds if and only if node n_1 has access to node n_2. Hence, a node can be accessed directly, or indirectly via a sequence of intermediate nodes. Now, suppose the task of a network administrator is to assign access rights by selecting a subset C_g of C to which some user will be granted access, whereby the user requires access to a set of nodes $C_a \subseteq C$ and is not permitted to access nodes from a set $C_f \subseteq C$ disjoint from C_a. Thus, the problem formally consists of selecting nodes $C_g \subseteq C$ such that every node in C_a is reachable from some node in C_g and no node in C_f is reachable from any node in C_g via edges in A.

In our problem setting, we assume the network is not known initially, but each node can be queried for the set of nodes it can access directly. For this, we use an external atom $\&hasAccess[nodes](n)$, which interfaces external network information, and outputs all nodes that can be accessed by some node in the extension of *nodes*. Accordingly, it evaluates to true for an output node n_2 wrt. an assignment $\mathbf{A}$ iff $\mathbf{T}nodes(n_1) \in \mathbf{A}$ for some $(n_1, n_2) \in A$. Moreover, we specify $D(\&hasAccess[nodes]) = \{\langle 2, 1 : \{n_1 \mid n_1 \to n_2 \in A\}, 1 : n_2\rangle \mid n_2 \in C\}$, i.e. there is a dependency of an output on an input node whenever the latter has access to the former. The HEX program in Fig. 2 with facts $domain(n)$ for $n \in C$, facts $node\_a(n)$ for $n \in C_a$, and facts $node\_f(n)$ for $n \in C_f$ encodes User Access Selection, where at most three nodes can be accessed directly.

First, we randomly generated networks with $N \in [10, 50]$ nodes, where each node has access to another node with probability $\frac{1}{2\times N}$ (cf. Table 1). As visible from the rightmost column in Table 1, this yields networks roughly half of which have no cyclic access relations and thus, dependency pruning can affect the number of required e-minimality checks. Next, we increased the access probability to $\frac{2}{N}$ (cf. Table 2). This effects that nearly all networks contain cycles, which allowed us to investigate the effect of pruning when this does not impact the need for an e-minimality check. Finally, we generated instances again with access probability $\frac{1}{2\times N}$, but removed all cyclic instances (cf. Table 3). The goal was to test instances where the e-minimality check can always be skipped, in order to ascertain the maximum speedup obtainable by pruning the dependency graph. The rightmost column in the tables shows the fraction of instances where the computer network contains a cycle.

Sequential Allocation of Indivisible Goods (SAIG). Next, we considered a problem from the area of *social choice*, namely dividing a set G of m items among two agents a_1 and a_2 by allowing them to pick items in specific sequences $\sigma = o_1 o_2 \ldots o_m \in \{a_1, a_2\}^m$ [53]. Each agent a_i has a linear preference order $>_i$ over G; and the utility of $g \in G$ for a_i is $u_i(g) = |\{g' \mid g >_i g' \in G\}|$. We assume that an agent always picks the remaining item with maximal utility. The goal is to find a

Table 3
Results for User Access Selection (all answer sets; no cycles).

#	All Answer Sets						#cyclic
	c-mod	**c-mod +io-dep**	**part**	**part +io-dep**	**min-part**	**min-part +io-dep**	
10	0.37 (0)	0.30 (0)	0.52 (0)	0.45 (0)	0.82 (0)	**0.29** (0)	0/10
15	1.48 (0)	**0.98** (0)	2.62 (0)	2.05 (0)	3.16 (0)	0.99 (0)	0/10
20	5.82 (0)	**3.28** (0)	14.11 (0)	12.51 (0)	10.98 (0)	3.37 (0)	0/10
25	42.02 (1)	**8.59** (0)	88.35 (1)	55.00 (0)	39.17 (0)	8.87 (0)	0/10
30	29.32 (0)	**18.83** (0)	124.64 (2)	115.18 (2)	84.21 (0)	18.88 (0)	0/10
35	73.73 (0)	**42.48** (0)	167.40 (3)	155.55 (3)	135.24 (3)	43.23 (0)	0/10
40	182.62 (2)	**100.21** (0)	238.41 (5)	209.27 (4)	189.40 (3)	103.89 (0)	0/10
45	226.40 (2)	**131.75** (1)	253.80 (5)	205.97 (5)	230.21 (3)	131.95 (1)	0/10
50	226.55 (5)	187.30 (2)	223.71 (5)	198.84 (3)	222.62 (5)	**179.50** (2)	0/10

Table 4
Results for Sequential Allocation of Indivisible Goods (all answer sets).

#	All Answer Sets					
	c-mod	**c-mod +io-dep**	**part**	**part +io-dep**	**min-part**	**min-part +io-dep**
3	**0.19** (0)	**0.19** (0)	0.25 (0)	0.24 (0)	0.38 (0)	**0.19** (0)
4	2.74 (0)	1.73 (0)	0.74 (0)	**0.64** (0)	2.54 (0)	1.72 (0)
5	300.00 (10)	78.28 (0)	152.33 (5)	**2.42** (0)	141.76 (1)	78.02 (0)
6	300.00 (10)	300.00 (10)	300.00 (10)	**8.22** (0)	300.00 (10)	300.00 (10)
7	300.00 (10)	300.00 (10)	300.00 (10)	**26.63** (0)	300.00 (10)	300.00 (10)
8	300.00 (10)	300.00 (10)	300.00 (10)	**89.97** (0)	300.00 (10)	300.00 (10)
9	300.00 (10)	300.00 (10)	300.00 (10)	**284.17** (4)	300.00 (10)	300.00 (10)
10	**300.00** (10)	**300.00** (10)	**300.00** (10)	**300.00** (10)	**300.00** (10)	**300.00** (10)

$$
\begin{aligned}
turn(a\_1, P) \vee turn(a\_2, P) &\leftarrow position(P).\\
picked(A, P, G) &\leftarrow \&pick[alreadyPicked](A, P, G),\\
&\qquad turn(A, P), item(G).\\
alreadyPicked(P, G) &\leftarrow position(P), position(P1),\\
&\qquad P1 < P, picked(\_, P1, G).\\
&\leftarrow \text{not } \&envyFree[picked]().
\end{aligned}
$$

Fig. 3. Sequential Allocation rules.

sequence σ resulting in an *envy-free* division of items, i.e. where no agent prefers the items of the other agent over its own items.

We use an external atom to obtain the choices of the agents, while their complete preferences are hidden, and a further one that checks whether an allocation is envy-free. The atom $\&pick[alreadyPicked](a_i, p, g)$ evaluates to true wrt. assignment $\mathbf{A}$ iff $p \in [1, m]$ and $g >_i g'$ for all g' such that $\mathbf{T}alreadyPicked(p-1, g) \notin \mathbf{A}$, where p represents the positions in a respective sequence. Furthermore, let

$$G(\mathbf{A}, i, j) = \sum_{g:\mathbf{T}picked(a_i,p,g)\in\mathbf{A}} u_j(g).$$

Then, the external atom $\&envyFree[picked]()$ is true iff $G(\mathbf{A}, 1, 1) < G(\mathbf{A}, 2, 1)$ and $G(\mathbf{A}, 2, 2) < G(\mathbf{A}, 1, 2)$. The encoding is shown in Fig. 4. Together with facts $position(p)$ and $item(g)$ for all $p, g \in [1, m]$, its answer sets encode all sequences that induce an envy-free allocation.

We specified the io-dependencies $D(\&pick[alreadyPicked]) = \{\langle 1, 1{:}\{p\}, 2{:}p+1\rangle \mid 1 \le p < m\}$, i.e. items already picked at a sequence position only depend on the previous positions. The io-dependencies eliminate all cyclic dependencies via external atoms in the instances; thus e-minimality checks can always be skipped. We tested instances with random preference orders and $N \in [3, 10]$ items (cf. Table 4).

Strategic Companies (SC). The problem consists in finding minimal sets of companies that jointly produce a fixed number of goods, whereby control relations between companies need to be taken into account [10]. We consider a setting similar to the one described by Leone et al. [56] and define a strategic companies problem as a four-tuple (C, G, P, O), where C is a set of m companies, G is a set of n goods, $P \subseteq G \times C \times C$ is a set of producer relations, and $O \subseteq C \times C \times C \times C$ is a set of control relations. Product g is produced by companies c_1 and c_2 if $(g, c_1, c_2) \in P$ and company c is jointly controlled by companies c_1, c_2 and c_3 if $(c, c_1, c_2, c_3) \in O$. A solution to (C, G, P, O), called a *strategic set*, is a set $C' \subseteq C$ such that

(1) for every $g \in G$ there is some $(g, c_1, c_2) \in P$ with $C' \cap \{c_1, c_2\} \neq \emptyset$,

Table 5
Results for strategic companies (all answer sets).

#	All Answer Sets					
	c-mod	**c-mod +io-dep**	**part**	**part +io-dep**	**min-part**	**min-part +io-dep**
10	0.15 (0)	**0.12** (0)	0.15 (0)	0.13 (0)	0.20 (0)	**0.12** (0)
15	**0.15** (0)	**0.15** (0)	0.22 (0)	0.21 (0)	0.54 (0)	0.16 (0)
20	0.25 (0)	0.25 (0)	0.37 (0)	0.34 (0)	1.22 (0)	**0.24** (0)
25	0.71 (0)	**0.58** (0)	0.94 (0)	0.74 (0)	4.42 (0)	0.60 (0)
30	1.97 (0)	**1.46** (0)	2.18 (0)	1.61 (0)	11.53 (0)	1.49 (0)
35	7.46 (0)	4.97 (0)	5.07 (0)	**3.44** (0)	39.33 (0)	5.10 (0)
40	22.89 (0)	14.16 (0)	15.70 (0)	**10.03** (0)	127.73 (0)	14.14 (0)
45	98.09 (0)	55.59 (0)	42.61 (0)	**24.57** (0)	280.40 (6)	55.81 (0)
50	134.50 (0)	80.95 (0)	87.68 (0)	**44.29** (0)	300.00 (10)	80.81 (0)
55	282.19 (9)	255.77 (2)	273.30 (6)	**163.05** (0)	300.00 (10)	256.53 (2)
60	300.00 (10)	300.00 (10)	300.00 (10)	**240.12** (2)	300.00 (10)	300.00 (10)

$$\begin{aligned} \mathit{strategic}(C1) \vee \mathit{strategic}(C2) &\leftarrow \mathit{produced\_by}(P, C1, C2). \\ \mathit{strategic}(C) &\leftarrow \&\mathit{controlled}[\mathit{strategic}](C), \\ &\quad\ \mathit{company}(C). \end{aligned}$$

Fig. 4. Strategic Companies rules.

(2) for every $(c, c_1, c_2, c_3) \in O$ where $\{c_1, c_2, c_3\} \subset C$ it holds that $c \in C'$,
(3) and C' is subset minimal wrt. conditions (1) and (2).

Intuitively, every product must be produced by some company in a strategic set, and the set must be closed under the control relation.

In our HEX encoding for computing strategic sets, shown in Fig. 3, we utilize an external atom $\&\mathit{controlled}[\mathit{strategic}](c)$ to retrieve all companies that are controlled by companies in a (candidate) strategic set. It evaluates to true wrt. assignment $\mathbf{A}$ iff $\mathbf{T}\mathit{strategic}(c_i) \in \mathbf{A}$ for some $(c, c_1, c_2, c_3) \in O$ and all $1 \leq i \leq 3$. Given a strategic companies problem (C, G, P, O), the answer sets of the rules in Fig. 3 together with facts $\mathit{produced\_by}(p, c_1, c_2)$ for each $(g, c_1, c_2) \in P$ and $\mathit{company}(c)$ for each $c \in C$ encode all strategic sets. For the ge-predicate $\&\mathit{controlled}[\mathit{strategic}]$, we specify the io-dependencies $D(\&\mathit{controlled}[\mathit{strategic}]) = \big\{\langle\{c_1, c_2, c_3 \mid (c, c_1, c_2, c_3) \in O\}, c\rangle \mid c \in C\big\}$.

We note that by employing the external atom for accessing information regarding which companies control other companies, our encoding differs from the standard formulation often used in ASP competitions for evaluating state-of-the-art ASP systems. In practice, accessing external information about the control relation may be necessary for the purpose of information hiding, i.e. in case only relevant information should be accessed without importing the complete graph. Information outsourcing may also be useful when the graph of the control relation is very large, or when additional external (pre-)processing of the information is necessary. Consequently, the variation of the strategic companies benchmark used here does not lend itself for a direct comparison with standard ASP solvers such as CLINGO, or ASP solvers with external source access that employ a semantics different from the one of DLVHEX. On the other hand, when no external sources are present in a program, DLVHEX forwards the complete computation to CLINGO such that for the classical strategic companies benchmark, there is no difference in performance.

We randomly generated strategic companies problems for testing, containing $N \in [10, 60]$ companies, $5 \times N$ producer relations in P and $N/2$ controls relations in O (cf. Table 5). Moreover, we assumed a total ordering $\succ$ of the elements in C and added the constraint that $(c, c_1, c_2, c_3) \in O$ only if $c \succ c_i$, for $1 \leq i \leq 3$, i.e. there is no cyclic control between companies. This, however, cannot be detected by analyzing the HEX program without taking the external source semantics into account. Accordingly, by exploiting the information provided in the form of io-dependencies, the solver can detect that the e-minimality check can be skipped in all cases, which would not be possible without dependency pruning.

5.2.3. Findings

We observe that when dependency graph pruning skips e-minimality checks, adding **+io-dep** significantly improves the running times for all instance sizes and independent from the configuration it is combined with (cf. Tables 1, 3, 4 and 5). Accordingly, the results support hypothesis (H1).

In Table 3, we observe an increased benefit of adding **+io-dep** due to the absence of cyclic instances. As expected, by testing single instances, we found that adding **+io-dep** reduces the running times roughly by the amount required for redundant e-minimality checks. In Table 2, only a negligible impact on the running time is visible when **+io-dep** is added. Since adding **+io-dep** has no advantage for cyclic instances, this shows that pruning the dependency graph yields no significant overhead and provides evidence for the correctness of hypothesis (H3).

Partial evaluation was only beneficial both in the model search and the e-minimality check for the SAIG benchmark, while only the configuration **part** exhibits a speedup in Tables 2 and 5. The reason for partial evaluation increasing the running time in all other cases is that the overhead that results from additional external calls did not outweigh the benefit in

terms of additional information gain. As predicted by hypothesis (H2), the speedup for **part**+**io-dep** in Table 4 is larger than for **min-part**+**io-dep** because **min-part** already reduces the running time required for e-minimality checks, while condition **part** needs to invest more time in the e-minimality check. The running times for **c-mod**+**io-dep** and **min-part**+**io-dep** are similar; this is expected as **min-part** only applies to the e-minimality check, which is skipped in both conditions. Moreover, adding +**io-dep** to configuration **min-part** results in a large speedup in case of the Strategic Companies benchmark because partial evaluation significantly increases the running time of the e-minimality check in this case, which however can always be skipped by leveraging the additional io-dependency information.

In summary, even though there is no clear winner among the conditions, adding +**io-dep** is suggestive as a default when io-dependencies can be specified as it often results in significant performance gains while not adding a large overhead otherwise.

5.2.4. Best-case scenario

While our experiments show that e-minimality check pruning clearly improves performance, gains are often moderate. This is partly due to the fact that the e-minimality check of DLVHEX is already highly optimized. For instance, properties of external sources such as monotonicity and functionality are exploited during the search for unfounded sets (cf. [66] for details), and observed input-output behavior of external sources is learned in the form of nogoods which can propagate during the main search as well as the e-minimality check. Nevertheless, even exponential speedups are theoretically possible and may occur in certain real-world settings. To illustrate this, we conclude this section by describing a generic best-case scenario which may appear in a range of practical use cases.

Consider the case where a partial configuration of some kind, represented by a set of items, is given, based on which a complete configuration that satisfies certain requirements should be computed. This could be modeled by the following simple rule:

$$selected(X) \leftarrow \&completeConfig[selected](X), item(X). \tag{5}$$

All available items are added by facts of the form $item(c)$, and the partial configuration by facts of the form $selected(c)$. Based on already selected items, the external atom outputs additional items which are required for completing the configuration. Note that more than one possible completion may exist.

Now, suppose that the output of the external atom can depend arbitrarily on the extension of the input predicate *selected*, i.e. the presence and absence of each input item can affect the presence of a given output item. For instance, item c may be added to a partial configuration if items a and b are already contained, but only if item d is not. As a result, for checking e-minimality of a completed configuration, it must be checked for each subset of the added items whether it results in a smaller model. At the same time, an exponential number of external calls can be saved when the e-minimality check can be skipped because the atom dependency graph computed wrt. a set of specified io-dependencies does not contain an e-cycle. Especially when the computation performed by the external source takes more time, this can have a drastic impact on the overall running time.

For a concrete example, consider route planning for a package delivery service. A partial configuration could represent a set of fixed stops, to which additional stops are to be added under certain conditions. For instance, if stops a and c are part of a route, stop b should also be added because it is located between a and c, but only if stop d is not part of the route as then adding b would constitute a detour. In this scenario, e-minimality checks could be skipped if the presence of additional stops would only depend on other stops that are located closer to the starting point of the route; this would impose an order over stops preventing cyclic dependencies.

Another use case would be diagnosing an illness based on symptoms, where a partial configuration represents a set of observed symptoms of a patient. Additional symptoms could be derived that are expected to be present under certain conditions, and the completed set of symptoms could be matched against the actual symptoms of the patient to determine a possible diagnosis. In this case, cyclic dependencies would be avoided when for instance symptoms can be categorized by their frequency, and less frequent symptoms are only added based on the presence or absence of more frequent symptoms.

6. Discussion

In this section, we consider related work, possible extensions and further applications of the results in this paper.

6.1. Related work

Dependencies are ubiquitous in *computer science*, and the use of graph structures in order to represent and express dependencies is popular; they are widely used in databases and information modeling, in programming languages, in software engineering etc.; the account given in the comparison to our work here is inherently incomplete, and moreover we concentrate on pointing out commonalities and differences rather than in-depth discussions.

Databases and information modeling. Specifically, in relational databases, dependency theory has been an important topic in data model design [62,1], with functional dependencies (which amount to Horn functions) at the core. The respective dependencies are at the generic, structural level, however, and not at the level of individual values as in our work.

The idea of exploiting information about dependencies between probabilistic variables constitutes the basis of *probabilistic graphical models* (*PGMs*) such as *Bayesian networks* and *Markov random fields* [54]. PGMs are used for representing compositional probabilistic models that combine simple probability distributions over variables into larger models [45]. The explicit specification of dependencies between variables in these models allows us to succinctly represent and to efficiently reason with complex probability distributions. Moreover, when the dependencies of a graphical model are acyclic, as in the case of *Bayesian networks*, probabilistic inference can be performed more efficiently. Similarly as above, these dependencies are at the conceptual level, while ours are at the data level and allow for expressing particular dependencies that might be hard to characterize in abstract terms. Furthermore, acyclicity of dependencies is not a requirement but allows for simpler evaluation.

Related to Bayesian nets and Markov random fields are *ceteribus paribus* (CP) networks [7], which are a popular formalism for expressing preference among solutions to a problem that is attribute-based. Informally, preferences are value-based where statements say that when solutions S and S' have the same values $v_1, \dots v_m$ of attributes $A_1, \dots, A_m$ then for a target attribute A a concrete value v is preferred over a value v'; for example, if the meal is fish, then white wine is preferred over red wine. The non-dominated solutions S are then selected, where S dominates S' if it can be transformed into the latter by flipping the attribute value of A if it violates a preference. Like our io-dependencies, CP-preferences are value-based and express local dependencies. However, they are forward-directed ("if–then") and for one particular value assignment to the attributes $A_1, \dots, A_m$, viz. that atoms $A_1 = v_1, \dots, A_m = v_m$ are true, while our io-dependencies are backward-directed ("only if") and range over all truth assignments to these atoms (which would be considered separately). Furthermore, CP-preferences do not mention output values explicitly; the latter may be tracked down along preferences, which however may be computationally expensive, given the intractability of core problems of CP-nets [47].

Programming languages and computation. In programming languages, dependencies in programs are expressed by control flow graphs [2] and data flow graphs [78]; in rule-based languages, dependencies are commonly expressed in syntactic terms from the head to the body of a rule, captured by the well-known dependency graph; this view may be further refined, as was done e.g. with the *argument dependency graph* for defining fragments of ASP programs with function symbols [12] that can be finitely grounded. The nodes of this graph are the argument positions $p[i]$ of predicates that occur in a given program, and edges $p[i] \to q[j]$ are put between nodes if in some rule $p(t_1, \dots, t_n)$ occurs in the head and $q(s_1, \dots, s_m)$ in the positive body, such that the terms t_i and s_j share some variable. Cycles among nodes are allowed, as long as this does not allow for building infinitely many ground instances of t_j; e.g., a cycle induced by the rules $p(X) \leftarrow q(X)$ and $q(X) \leftarrow p(X)$ would be admitted. The emerging class of *finite domain* programs [12] has been later generalized to *argument restricted* programs [58]. In our work, we aim at foundedness, which is not an issue for the goal of program grounding as pursued in [12,58].

Input/output dependencies play an important role in the *component-based design* of computation models, where interfaces constrain the influence relations between inputs and outputs of components using so-called *input assumptions* and *output guarantees* [20]. For instance, mutual blocking of processes in synchronous component models can be prevented by ruling out certain compositions based on semantically or syntactically specified dependencies [17]. In this context, types can be used to make dependencies transparent and to prevent circular input/output dependencies between components. Our notion of dependencies is different in that no specific purpose is a priori intended, such as controlling and preventing cyclicity; on the contrary, the dependencies serve to single out (potential) cyclic cases which are dealt with by the answer set semantics; furthermore, the dependencies are at the level of particular data and not a higher level of abstraction. However, abstract (generic) dependencies that can be instantiated to the data level are conceivable.

Constraint solving and logic variable dependencies. For solving constraint satisfaction problems, constraint propagation builds on dependence information between variables $X_1, \dots, X_n$ which are the nodes of a (hyper)graph whose (hyper)edges link variables that co-occur in a constraint, cf. [5]; assigning a value to a variable will reduce the set of possible values for its neighbors, akin to literal propagation in the well-known DPLL-procedure for SAT. In a simple HEX encoding, we can regard a constraint $c(\vec{Y})$, where $\vec{Y} = X_{i_1}, \dots X_{i_k}$, as an external atom $\&con[val](\vec{Y})$ that filters from given value assignments $val(c_1, \dots, c_n)$ those restricted to $\vec{Y}$ such that $con(c_{i_1}, \dots, c_{i_k})$ holds; io-dependencies for $\&conc[val]$ aid in pruning guesses val for (sets of) solutions, whose correctness is checked with ASP constraints $\leftarrow val(\vec{X}), \text{not } \&con[val](\vec{Y})$. Such a naive encoding may however be more of theoretical interest than practically competitive; whether constraint propagation techniques can be exploited for advanced io-dependency processing is an interesting subject for future research.

A different use of dependency information in constraint solving was pursued in [61], where functional and finiteness dependencies have been generalized with constraints. In a similar vein, dependencies between variables in formalisms with more expressive quantification have been considered in quantified Boolean formulas [3] and in quantified constraint satisfaction problems [69]. There, explicit dependency information among quantified variables is given in syntactic form, similar as for *Henkin quantifiers* in first-order logic; dependency schemes [70] aim in this context at detecting spurious dependencies in dependency-aware QBF solving, and are in this sense related to dependency pruning by io-dependencies. In independence-friendly logic [48], functional independence of a value from a set of variables can be expressed (leading beyond first-order logic); a number of related logical systems have been proposed. These logics are thus in the tradition of structural dependence as modeling languages in the database context are (see above), while our work aims at the specific instance (value) level.

Software engineering. In software engineering, different kinds of dependencies are considered, such as dependencies in the process of software development, comprising knowledge, process and resource dependencies [74], or related to the latter, dependencies among modular components in a software architecture, where more dependence is assumed to allow for less independent development [46]. In deploying software, package management is plagued with vicious cycles, version conflicts, etc.; notably, ASP has been fruitfully used for escaping this "dependency hell" in the Linux world [76,40]. As in our work, dealing with cyclic dependencies is important, yet the analysis is at a coarse level. Gebser et al. [40] model packages with version numbers and property descriptions about conflicts, dependence, recommendation and providence regarding other packages. If the latter were available via external atoms, io-dependencies would help to prune relationships and avoid cycles; native ASP encodings from factual representations as given in [40] will however be more effective.

ASP with external access. Our approach is related to *domain independence* techniques in [22], where external atoms are evaluated wrt. subsets of the domain while correct outputs are retained. This is similar to our notions of compliant atoms and faithfulness. However, our io-dependencies are more general because in [22], only disjoint domain partitions for external inputs were considered, and no dependencies for argument positions were used. Another important difference is that the approach of Eiter et al. [22] employs dependencies for *program splitting* as in [60], while we aim at detecting redundant e-minimality checks. Furthermore, they did not analyze the costs of generating dependencies.

Apart from HEX, there are several other approaches that integrate external theories into declarative problem solving, such as CLINGO [37], *SMT* [4] and *Constraint-ASP* [57]. However, to the best of our knowledge, external minimality checking has not yet been considered there. Nevertheless, our technique could also be employed directly by related rule-based formalisms if minimality involving external theories is required. Moreover, cyclic support may arise from *external propagators*, e.g. in the WASP solver [19,18], where our approach could be applied as well.

For example, in case of the CLINGO system, the external atom $\&closeTo[city](X)$ from the example in Section 1 can be modeled by a theory atom $\&closeTo(X)$ available in CLINGO 5. Then, the same semantics as for the external atom of HEX can be defined for the theory atom, i.e. that $\&closeTo(X)$ is true for a city X if and only if $city(Y)$ is true for some city Y which is located close to city X. However, due to a difference in the semantics employed by CLINGO and HEX, respectively, the spurious answer set containing both $city(bratislava)$ and $city(vienna)$ (cf. the example in Section 1) would be returned by CLINGO. Due to the different semantics, CLINGO does not need to apply an additional e-minimality check; but a check similar to the one performed by DLVHEX could be added to avoid cyclic support involving theory atoms.

Note that in contrast to the external atoms of HEX, theory atoms in CLINGO do not have an input list of constants and predicate names by default. However, in theory, a scope of atoms is also associated with each interfaced theory, which may include ordinary as well as theory atoms. In practice, a theory propagator in CLINGO can set up watches on program literals to be notified about new truth value assignments. Accordingly, theory atoms could be annotated with names of predicates on whose extension their evaluation depends, and sets of compliant atoms could be computed based on io-dependencies as in the case of HEX. In turn, the presence of e-cycles in the corresponding atom dependency graph involving theory atoms instead of external atoms could be checked and an additional e-minimality check could be avoided whenever no e-cycle exists. Based on this adaptation of our contributions to the context of CLINGO, achieving similar performance gains as shown for DLVHEX should be possible.

6.2. Extensions

The approach presented can be extended in several ways; we briefly address two of them.

Io-dependency learning. To obtain dependencies for an external atom, one may employ heuristics and/or sampling techniques in order to generate candidate dependencies which then can be tested for faithfulness and be optimized. This may be done ad hoc, but will be costly in general; more promising seems to be offline computation (for a fixed external source), such that the io-dependencies found can be shared between different programs. Moreover, online optimization of such io-dependencies is possible when further information (e.g., from envelope computation or nogoods learned during the model search) becomes available. For instance, suppose the external atom $\&suc[node](c)$ from Example 11 is evaluated during HEX solving and the nogoods $\{\mathbf{T}node(d), \mathbf{F}\&suc[node](c)\}$ and $\{\mathbf{F}node(d), \mathbf{T}\&suc[node](c)\}$ are learned. Then, it can be inferred that only the truth value of the atom $node(d)$ can influence the value of $\&suc[node](c)$, and this information can be exploited for further pruning of the atom dependency graph on-the-fly during solving.

Generalized dependencies. In this work, we have considered elementary dependencies among constants occurring at single input and output positions. A natural extension is to consider dependencies that involve constants at multiple positions.

For example, consider an external atom such as $\&child[person](FN, LN)$, which outputs the names FN, LN of the children of each person that is in the given binary input predicate $person(FN, LN)$ by accessing an external source. That Joe Doe can be only the child of Ann Doe and Bob Doe may be expressed by a dependency $\delta_1 = \langle 1, \{(1{:}ann, 2{:}doe), (1{:}bob, 2{:}doe)\}, (1{:}joe, 2{:}doe)\rangle$, in a format generalizing the one from above, where the tuples $(1{:}ann, 2{:}doe)$ and $(1{:}joe, 2{:}doe)$ state the compliant atoms $person(ann, doe)$ and $person(bob, doe)$ for the output (joe, doe); that Joe Doe can only have Does as parents could be expressed by $\delta_2 = \langle 1, \{(2{:}doe)\}, (1{:}joe, 2{:}doe)\rangle$, which induces compliant atoms $a = person(c, doe)$ for any constant c.

More generally, also dependencies that involve constants across input predicates can be imagined; e.g. if in the previous example only children having a parent with first name in a second input *fname* should be output. The dependency above may then be altered to something like $\delta_3 = \langle\{[1 : (1{:}ann, 2{:}doe), 2 : (1{:}ann)],\ [1 : (1{:}bob, 2{:}doe), 2 : (1{:}bob)]\},\ (1{:}joe, 2{:}doe)\rangle$, where $[1 : (1{:}ann, 2{:}doe), 2 : (1{:}ann)]$ informally expresses the compliant atoms $person(ann, doe)$, $fname(ann)$, and similarly $[1 : (1{:}bob, 2{:}doe), 2 : (1{:}bob)]$ the compliant atoms $person(bob, doe)$, $fname(bob)$.

In the dependency graph, these generalized io-dependencies would induce external dependencies from compliant atoms as usual; e.g. δ_1 would for a rule $p(joe, doe) \leftarrow \&child[person](joe, doe)$ give rise to the external dependency $p(joe, doe) \rightarrow_e a$, for $a = person(ann, doe)$, $person(bob, doe)$.

In order to exploit the full potential of cross input io-dependencies like δ_3, a more expressive kind of dependency graph would be needed that can distinguish the joint evaluation of $person(ann, doe)$ and $fname(anne)$ from the one of $person(bob, doe)$ and $fname(bob)$. To this end, and-or graphs may be used and the notion of cyclicity refined, at the price of more expensive dependency checking (which increases from NLSPACE to PTIME, as reachability in and-or graphs is PTIME-complete, cf. [50]).

6.3. Further applications in program evaluation

While we have concentrated on exploiting io-dependencies for e-minimality checking, such dependency information is also useful for improving other tasks of the HEX program evaluation machinery.

Grounding. Following the traditional grounding-&-solving architecture of ASP solvers, the first step in the evaluation of a HEX program Π is to compute an equivalent (finite) grounding of Π resp. parts of it as mentioned in Section 2.2. In this process, external atoms are evaluated repeatedly in order to obtain new values that do not yet occur in the incrementally built grounding. By respecting io-dependencies, the input of an external atom may be restricted to the compliant atoms (those on which the output values depend) already in the grounding phase. Furthermore, caching techniques may help to decrease the number of external evaluations when for a particular external atom input multiple resp. all output values are computed.

Solving. Similarly as during grounding, also in the step where the equivalent ground program is evaluated, io-dependencies may be used to restrict the atom input to relevant atoms. Such a restriction can be fruitfully used in partial evaluation, where the external evaluation is attempted with an incomplete input, i.e., under an assignment $\mathbf{A}$ which leaves the truth value of some atoms undefined [28]. By having irrelevant atoms pruned, the final value of an external atom under any completion of $\mathbf{A}$ to a complete assignment may be determined quicker.

Behavior learning. An important feature of the DLVHEX solver is *external behavior learning* [28], which aims at learning nogoods for the external function access. By limiting oracle calls to compliant input atoms, the number of external calls during HEX-evaluation could potentially be reduced significantly.

Summarizing, io-dependencies complement the tool-box of semantic properties of external atoms at an abstract level (finiteness, monotonicity, functionality etc.), by adding properties for concrete elements, which can be exploited at multiple points of HEX program evaluation.

7. Conclusion

The extension of ASP with access to external sources of information widens the possible range of applications, but at the same time poses challenges for the efficient evaluation of ASP programs. While an API-style interface mechanism is beneficial for information hiding and privacy, it makes the computation of answer sets in a declarative problem solving setting difficult. In this paper, we have considered semantic dependency information between input and output of external sources as an approach for optimization. Specifically, we have introduced io-dependencies to formalize semantic dependencies over external atoms. Thanks to this notion, the black-box nature of external atoms has been softened with a more accurate approximation of the real dependencies than what was previously available. As a consequence, more (expensive) e-minimality checks for answer set candidates can be skipped in the evaluation of programs with external source access, which proved to be beneficial in practice as shown by experimental results. Furthermore, we have also established properties for checking and for optimizing io-dependencies. The latter can be exploited for the automatic construction of faithful dependency sets that are tight, i.e., those which introduce a smallest set of edges in the dependency graph of the program. In turn, tight sets of faithful io-dependencies allow for a maximal pruning of e-minimality checks during HEX evaluation. While faithfulness checking of io-dependencies is intractable in general, we have also identified cases where the costs can be reduced and where the faithfulness check is feasible in polynomial time. Our results on io-dependencies may be used for other tasks in evaluating HEX programs, and they may be used in the context of other extensions of ASP with external access as well.

Outlook. In this article, we have provided some notions for the use of semantic dependency information in ASP with external information access. This work can be continued and the study extended in future research in several directions. One

direction is to explore other envelopes apart from the guessing and the Horn envelopes. In fact, the latter may be just a crude over-approximation of the best envelope possible; to this end, further refinements in trade-off with computational efforts may be designed.

Another research direction is to explore more general notions of dependencies, such as dependencies among tuples of constants as discussed in Section 6.2. Similar to the use of envelopes, it will in this context be intriguing to find a good balance between the expressiveness of the language for dependencies and the computational cost in terms of space and time to deploy specifications in it.

Finally, an important direction is to develop tools for dependency management and to integrate them into ASP development environments as, e.g., discussed in [59]. Core services that need to be supported are io-dependency specification, testing and optimization, while convenient editors and visualization tools are important to make the ASP approach more user friendly and accessible.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been supported by the Austrian Science Fund (FWF) via the projects P27730 and W1255-N23. We would like to thank the reviewers for pointing out corrections and their helpful and constructive comments to improve the presentation of this work.

Appendix A. Emulating negation and disjunction

We provide here a proof of the claim in Section 4.2.1. Let $rew_{\text{not},\vee}(\Pi)$ be the program that results from a HEX program Π by rewriting negation 'not' and disjunction '$\vee$' using external atoms as described there (technically, for each $\vec{X} = X_1, \ldots X_k$ of length k we use designated external predicates $\&naf^k$ and $\&disj^k_l$, but omit the details for simplicity).

Proposition 14. *Let Π be a* HEX *program in which all disjunctions are of the form $p(\vec{X}) \vee q(\vec{X})$, where p and q are distinct predicates. Then, an assignment $\mathbf{A}$ is an answer set of Π iff $\mathbf{A}$ is an answer set of $rew_{\text{not},\vee}(\Pi)$.*

Proof. For any assignment $\mathbf{A}$, by definition (*) $\mathbf{A} \models \text{not } p(\vec{c})$ iff $\mathbf{A} \models \&naf[p](\vec{c})$ for any ground atom $p(\vec{c})$, and for any ground disjunction $p(\vec{c}) \vee q(\vec{c})$ we have (**) $\mathbf{A} \not\models p(\vec{c}) \vee q(\vec{c})$ iff $\mathbf{A} \not\models \&dis_l[p,q](c)$ and $\mathbf{A} \not\models \&dis_l[q,p](c)$.

Consider any rule $r \in grnd(\Pi)$, and let r' be a rule obtained from r by rewriting. If (a) r is non-disjunctive, then (*) implies that $r' \in frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$ iff $r \in f\Pi^{\mathbf{A}}$. If (b) r is disjunctive, either (b.1) $H(r') = p(\vec{c})$ and then $r' \in frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$ iff $r \in f\Pi^{\mathbf{A}}$ and ($\mathbf{A} \models p(\vec{c})$ or $\mathbf{A} \not\models q(\vec{c})$), or (b.2) $H(r') = q(\vec{c})$ and then $r' \in frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$ iff $r \in f\Pi^{\mathbf{A}}$ and ($\mathbf{A} \not\models p(\vec{c})$ or $\mathbf{A} \models q(\vec{c})$).

($\Rightarrow$) Consider any answer set $\mathbf{A}$ of Π. We first show that $\mathbf{A}$ is a model of $frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$. Towards a contradiction, suppose $r \in grnd(\Pi)$ has some rewriting r' such that $r' \in frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$ and $\mathbf{A}$ violates r', i.e., $\mathbf{A} \models B(r')$ and $\mathbf{A} \not\models H(r')$. From (a), (b.1), and (b.2) we conclude that $r \in f\Pi^{\mathbf{A}}$, and thus $\mathbf{A} \models B(r)$. In case (a), we obtain $\mathbf{A} \models H(r)$, and thus $\mathbf{A} \models H(r')$ which is a contradiction. In case (b.1), as $\mathbf{A} \models H(r)$, i.e., $\mathbf{A} \models p(\vec{c}) \vee q(\vec{c})$, and $\mathbf{A} \models \&dis_l[p,q](c)$, i.e., $\mathbf{A} \models p(\vec{c}) \vee \neg q(\vec{c})$, we obtain $\mathbf{A} \models p(\vec{c})$, which means again $\mathbf{A} \models H(r')$ and a contradiction; similarly in case (b.2). Thus $\mathbf{A}$ is a model of $frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$.

To show that $\mathbf{A}$ is minimal, assume $\mathbf{A}' < \mathbf{A}$ is a model of $frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$. We show that then $\mathbf{A}'$ is a model of $f\Pi^{\mathbf{A}}$, which contradicts the minimality of $\mathbf{A}$. If $\mathbf{A}'$ were not a model of $f\Pi^{\mathbf{A}}$, some rule $r \in f\Pi^{\mathbf{A}}$ exists such that $\mathbf{A}' \models B(r)$ and $\mathbf{A}' \not\models H(r)$. Let r' be obtained from r by rewriting. In case (a), we have $r' \in frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$ and from (*) that $\mathbf{A}' \models B(r')$. Hence $\mathbf{A}' \models H(r')$ which implies $\mathbf{A}' \models H(r)$, a contradiction. In case (b), as $\mathbf{A} \models H(r)$ we conclude from (**) that either in case (b.1) or in case (b.2) we have $r' \in frew_{\text{not},\vee}(\Pi)^{\mathbf{A}}$. As $\mathbf{A}' \models B(r')$, it follows that $\mathbf{A}' \models H(r')$ and thus $\mathbf{A}' \models H(r)$, which is again a contradiction. Thus, $\mathbf{A}'$ is a model of $f\Pi^{\mathbf{A}}$, which is a contradiction.

($\Leftarrow$) The proof that every answer set $\mathbf{A}$ of $rew_{\text{not},\vee}(\Pi)$ is an answer set of Π is similar. We first show that $\mathbf{A}$ is a model of $f\Pi^{\mathbf{A}}$. If not, some rule $r \in f\Pi^{\mathbf{A}}$ exists such that $\mathbf{A} \models B(r)$ and $\mathbf{A} \not\models H(r)$. Let r' be a rewriting of r. In case (a), we have $r' \in frew_{\text{not},\vee}(\Pi)$ and by (*) that $\mathbf{A} \models H(r')$, thus $\mathbf{A} \models H(r)$, which is a contradiction. In case (b), if $H(r') = p(\vec{c})$ we have by (b.1) that $r' \in frew_{\text{not},\vee}(\Pi)$ and obtain $\mathbf{A} \models H(r')$, thus $\mathbf{A} \models H(r)$, again a contradiction. Analogous for $H(r') = q(\vec{c})$. Thus $\mathbf{A}$ is a model of $frew_{\text{not},\vee}(\Pi)$.

To show that $\mathbf{A}$ is minimal, assume that $\mathbf{A}' < \mathbf{A}$ is a model of $f\Pi^{\mathbf{A}}$. We show that then $\mathbf{A}'$ is a model of $frew_{\text{not},\vee}(\Pi)$, which contradicts the minimality of $\mathbf{A}$. If $\mathbf{A}'$ were not a model of $frew_{\text{not},\vee}(\Pi)$, some rule $r' \in frew_{\text{not},\vee}(\Pi)$ exists such that $\mathbf{A}' \models B(r')$ and $\mathbf{A}' \models H(r')$. If r' is the rewriting of a disjunction-free rule r, we obtain from (a) that $r \in f\Pi^{\mathbf{A}}$. From (*), we conclude $\mathbf{A}' \models B(r)$ and thus $\mathbf{A}' \models H(r)$, which implies $\mathbf{A}' \models H(r')$, which is a contradiction. If r' stems from a disjunctive rule r, we obtain from (b.1) and (b.2) that $r \in f\Pi^{\mathbf{A}}$. As $\mathbf{A}' \models B(r')$ and $B(r) \subseteq B(r')$, we have $\mathbf{A}' \models B(r)$, and hence $\mathbf{A}' \models H(r)$. Now if $\mathbf{A}' \models p(\vec{c})$, it follows that $H(r') \neq p(\vec{c})$ and thus $H(r') = q(\vec{c})$ and $\mathbf{A}' \not\models q(\vec{c})$ must hold; hence, $\mathbf{A}' \not\models \&dis_l[q,p](c)$

which however contradicts that $\mathbf{A}' \models B(r')$. If otherwise $\mathbf{A}' \models q(\vec{c})$, we similarly derive a contradiction to $\mathbf{A}' \models B(r')$. Thus, $\mathbf{A}'$ must be a model of $frew_{\mathrm{not},\vee}(\Pi)$, which is a contradiction. This proves the result. □

We remark that the rewriting $rew_{\mathrm{not},\vee}(\Pi)$ can be extended (using auxiliary predicates) to programs with arbitrary disjunctions in the head; however, the class of programs covered already includes worst-case complexity instances for disjunctive programs with bounded predicate arities.

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.
[2] F.E. Allen, Control flow analysis, SIGPLAN Not. 5 (7) (Jul. 1970) 1–19, http://doi.acm.org/10.1145/390013.808479.
[3] V. Balabanov, H.K. Chiang, J.R. Jiang, Henkin quantifiers and boolean formulae: a certification perspective of DQBF, Theor. Comput. Sci. 523 (2014) 86–100, https://doi.org/10.1016/j.tcs.2013.12.020.
[4] C.W. Barrett, R. Sebastiani, S.A. Seshia, C. Tinelli, Satisfiability modulo theories, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, in: Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009, pp. 825–885.
[5] C. Bessiere, Constraint propagation, in: F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, in: Foundations of Artificial Intelligence, vol. 2, Elsevier, 2006, pp. 29–83.
[6] B. Bogaerts, E. Erdem, A. Harrison, Guest editorial: special issue on answer set programming and other computing paradigms, Ann. Math. Artif. Intell. 86 (1–3) (2019) 1–2, https://doi.org/10.1007/s10472-019-09634-w.
[7] C. Boutilier, R.I. Brafman, C. Domshlak, H.H. Hoos, D. Poole, CP-nets: a tool for representing and reasoning with conditional ceteris paribus preference statements, J. Artif. Intell. Res. 21 (2004) 135–191, https://doi.org/10.1613/jair.1234.
[8] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (12) (2011) 92–103, https://doi.org/10.1145/2043174.2043195.
[9] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming: an introduction to the special issue, AI Mag. 37 (3) (2016) 5–6, http://www.aaai.org/ojs/index.php/aimagazine/article/view/2669.
[10] M. Cadoli, T. Eiter, G. Gottlob, Default logic as a query language, IEEE Trans. Knowl. Data Eng. 9 (3) (1997) 448–463, https://doi.org/10.1109/69.599933.
[11] F. Calimeri, S. Cozza, G. Ianni, External sources of knowledge and value invention in logic programming, Ann. Math. Artif. Intell. 50 (3–4) (2007) 333–361, https://doi.org/10.1007/s10472-007-9076-z.
[12] F. Calimeri, S. Cozza, G. Ianni, N. Leone, Computable functions in ASP: theory and implementation, in: M.G. de la Banda, E. Pontelli (Eds.), Logic Programming, Proceedings of 24th International Conference, ICLP 2008, Udine, Italy, December 9-13, 2008, in: Lecture Notes in Computer Science, vol. 5366, Springer, 2008, pp. 407–424.
[13] F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci, A. Wimmer, Angry-HEX: an artificial player for angry birds based on declarative knowledge bases, IEEE Trans. Comput. Intell. AI Games 8 (2) (2016) 128–139, https://doi.org/10.1109/TCIAIG.2015.2509600.
[14] F. Calimeri, D. Fuscà, S. Germano, S. Perri, J. Zangari, Fostering the use of declarative formalisms for real-world applications: the EmbASP framework, New Gener. Comput. 37 (1) (2019) 29–65, https://doi.org/10.1007/s00354-018-0046-2.
[15] A. Chandra, P. Merlin, Optimal implementation of conjunctive queries in relational databases, in: Proceedings of the Ninth ACM Symposium on the Theory of Computing, STOC-77, Boulder, Colorado, 1977, pp. 77–89.
[16] C. Chekuri, A. Rajaraman, Conjunctive query containment revisited, in: F.N. Afrati, P.G. Kolaitis (Eds.), Database Theory - ICDT '97, Proceedings of 6th International Conference, Delphi, Greece, January 8–10, 1997, in: Lecture Notes in Computer Science, vol. 1186, Springer, 1997, pp. 56–70.
[17] L. de Alfaro, T.A. Henzinger, F.Y.C. Mang, The control of synchronous systems, in: C. Palamidessi (Ed.), CONCUR 2000 - Concurrency Theory, Proceedings of 11th International Conference, University Park, PA, USA, August 22-25, 2000, in: Lecture Notes in Computer Science., vol. 1877, Springer, 2000, pp. 458–473.
[18] C. Dodaro, F. Ricca, The external interface for extending WASP, TPLP (2018) 1–24, https://doi.org/10.1017/S1471068418000558.
[19] C. Dodaro, F. Ricca, P. Schüller, External propagators in WASP: preliminary report, in: S. Bistarelli, A. Formisano, M. Maratea (Eds.), Proceedings of the 23rd RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, RCRA 2016, Genova, Italy, November 28, 2016, in: CEUR Workshop Proceedings, vol. 1745, CEUR-WS.org, 2016, pp. 1–9, http://ceur-ws.org/Vol-1745/paper1.pdf.
[20] L. Doyen, T.A. Henzinger, B. Jobstmann, T. Petrov, Interface theories with component reuse, in: L. de Alfaro, J. Palsberg (Eds.), Proceedings of the 8th ACM & IEEE International Conference on Embedded Software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008, ACM, 2008, pp. 79–88.
[21] T. Eiter, W. Faber, M. Fink, S. Woltran, Complexity results for answer set programming with bounded predicate arities and implications, Ann. Math. Artif. Intell. 51 (2–4) (2007) 123–165, https://doi.org/10.1007/s10472-008-9086-5.
[22] T. Eiter, M. Fink, T. Krennwallner, Decomposition of declarative knowledge bases with external functions, in: C. Boutilier (Ed.), IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, 2009, pp. 752–758, http://ijcai.org/Proceedings/09/Papers/130.pdf.
[23] T. Eiter, M. Fink, T. Krennwallner, C. Redl, Domain expansion for ASP-programs with external sources, Artif. Intell. 233 (2016) 84–121, https://doi.org/10.1016/j.artint.2016.01.003.
[24] T. Eiter, M. Fink, T. Krennwallner, C. Redl, P. Schüller, Efficient hex-program evaluation based on unfounded sets, J. Artif. Intell. Res. 49 (2014) 269–321, https://doi.org/10.1613/jair.4175.
[25] T. Eiter, S. Germano, G. Ianni, T. Kaminski, C. Redl, P. Schüller, A. Weinzierl, The DLVHEX system, Künstl. Intell. 32 (2–3) (2018) 187–189, https://doi.org/10.1007/s13218-018-0535-y.
[26] T. Eiter, G. Gottlob, H. Mannila, Disjunctive datalog, ACM Trans. Database Syst. 22 (3) (1997) 364–418, https://doi.org/10.1145/261124.261126.
[27] T. Eiter, T. Kaminski, C. Redl, P. Schüller, A. Weinzierl, Answer set programming with external source access, in: G. Ianni, D. Lembo, L.E. Bertossi, W. Faber, B. Glimm, G. Gottlob, S. Staab (Eds.), Reasoning Web. Semantic Interoperability on the Web - 13th International Summer School 2017, London, UK, July 7-11, 2017, Tutorial Lectures, in: Lecture Notes in Computer Science, vol. 10370, Springer, 2017, pp. 204–275.
[28] T. Eiter, T. Kaminski, C. Redl, A. Weinzierl, Exploiting partial assignments for efficient evaluation of answer set programs with external source access, J. Artif. Intell. Res. 62 (2018) 665–727, https://doi.org/10.1613/jair.1.11221.
[29] T. Eiter, C. Redl, P. Schüller, Problem solving using the HEX family, in: C. Beierle, G. Brewka, M. Thimm (Eds.), Computational Models of Rationality, Essays dedicated to Gabriele Kern-Isberner on the occasion of her 60th birthday, College Publications, 2016, pp. 150–174.
[30] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (3) (2016) 53–68, http://www.aaai.org/ojs/index.php/aimagazine/article/view/2678.
[31] E. Erdem, U. Öztok, Generating explanations for biomedical queries, Theory Pract. Log. Program. 15 (1) (2015) 35–78, https://doi.org/10.1017/S1471068413000598.
[32] E. Erdem, V. Patoglu, Applications of ASP in robotics, Künstl. Intell. 32 (2–3) (2018) 143–149, https://doi.org/10.1007/s13218-018-0544-x.

[33] W. Faber, G. Pfeifer, N. Leone, Semantics and complexity of recursive aggregates in answer set programming, Artif. Intell. 175 (1) (2011) 278–298, https://doi.org/10.1016/j.artint.2010.04.002.
[34] A.A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E.C. Teppan, Industrial applications of answer set programming, Künstl. Intell. 32 (2–3) (2018) 165–176, https://doi.org/10.1007/s13218-018-0548-6.
[35] O. Febbraro, N. Leone, G. Grasso, F. Ricca, JASP: a framework for integrating answer set programming with java, in: G. Brewka, T. Eiter, S.A. McIlraith (Eds.), Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012, AAAI Press, 2012, pp. 541–551, http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4520.
[36] D. Fuscà, S. Germano, J. Zangari, M. Anastasio, F. Calimeri, S. Perri, A framework for easing the development of applications embedding answer set programming, in: J. Cheney, G. Vidal (Eds.), Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016, ACM, 2016, pp. 38–49.
[37] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with Clingo 5, in: M. Carro, A. King, N. Saeedloei, M.D. Vos (Eds.), Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA, in: OASICS, vol. 52, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 2:1–2:15.
[38] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2012.
[39] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Clingo = ASP + control, Preliminary report, 2014, CoRR abs/1405.3694, http://arxiv.org/abs/1405.3694.
[40] M. Gebser, R. Kaminski, T. Schaub, aspcud: a Linux package configuration tool based on answer set programming, in: C. Drescher, I. Lynce, R. Treinen (Eds.), Proceedings Second Workshop on Logics for Component Configuration, LoCoCo 2011, Perugia, Italy, 12th September 2011, in: EPTCS, vol. 65, 2011, pp. 12–25.
[41] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, M.T. Schneider, Potassco: the Potsdam answer set solving collection, AI Commun. 24 (2) (2011) 107–124, https://doi.org/10.3233/AIC-2011-0491.
[42] M. Gelfond, Y. Kahl (Eds.), Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach, Cambridge University Press, 2014.
[43] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: R.A. Kowalski, K.A. Bowen (Eds.), Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes), MIT Press, 1988, pp. 1070–1080.
[44] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (3/4) (1991) 365–386, https://doi.org/10.1007/BF03037169.
[45] Z. Ghahramani, Probabilistic machine learning and artificial intelligence, Nature 521 (7553) (2015) 452–459, https://doi.org/10.1038/nature14541.
[46] C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering, 2nd ed., Prentice Hall, 2002.
[47] J. Goldsmith, J. Lang, M. Truszczynski, N. Wilson, The computational complexity of dominance and consistency in CP-nets, J. Artif. Intell. Res. 33 (2008) 403–432, https://doi.org/10.1613/jair.2627.
[48] J. Hintikka, G. Sandu, Informational independence as a semantical phenomenon, in: Fenstad, J.E. et al. (Eds.), Logic, Methodology and Philosophy of Science VIII, fourth edition, in: Studies in Logic and the Foundations of Mathematics, vol. 126, North-Holland Publishing Company, 1989, pp. 571–589.
[49] R. Hoehndorf, F. Loebe, J. Kelso, H. Herre, Representing default knowledge in biomedical ontologies: application to the integration of anatomy and phenotype ontologies, BMC Bioinform. 8 (2007), https://doi.org/10.1186/1471-2105-8-377.
[50] N. Immerman, Number of quantifiers is better than number of tape cells, J. Comput. Syst. Sci. 22 (3) (1981) 384–406, https://doi.org/10.1016/0022-0000(81)90039-8.
[51] T. Janhunen, G. Liu, I. Niemelä, Tight integration of non-ground answer set programming and satisfiability modulo theories, in: P. Cabalar, D. Mitchell, D. Pearce, E. Ternovska (Eds.), Informal Proceedings of the 1st Workshop on Grounding and Transformations for Theories with Variables, GTTV'11, LPNMR, Vancouver, BC, Canada May 16th, 2011, 2013, pp. 1–14, available online at http://www.dc.fi.udc.es/GTTV11/GTTV-Proc.pdf.
[52] T. Janhunen, E. Oikarinen, H. Tompits, S. Woltran, Modularity aspects of disjunctive stable models, J. Artif. Intell. Res. 35 (2009) 813–857, https://doi.org/10.1613/jair.2810.
[53] T. Kalinowski, N. Narodytska, T. Walsh, L. Xia, Strategic behavior when allocating indivisible goods sequentially, in: M. desJardins, M.L. Littman (Eds.), Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, Bellevue, Washington, USA, July 14-18, 2013, AAAI Press, 2013, http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6482.
[54] D. Koller, N. Friedman, Probabilistic Graphical Models - Principles and Techniques, MIT Press, 2009, http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11886.
[55] J. Lee, Y. Meng, Answer set programming modulo theories and reasoning about continuous changes, in: F. Rossi (Ed.), IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013, IJCAI/AAAI, 2013, pp. 990–996, http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6895.
[56] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (3) (2006) 499–562, https://doi.org/10.1145/1149114.1149117.
[57] Y. Lierler, Relating constraint answer set programming languages and algorithms, Artif. Intell. 207 (2014) 1–22, https://doi.org/10.1016/j.artint.2013.10.004.
[58] Y. Lierler, V. Lifschitz, One more decidable class of finitely ground programs, in: P.M. Hill, D.S. Warren (Eds.), Logic Programming, Proceedings of 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009, in: Lecture Notes in Computer Science, vol. 5649, Springer, 2009, pp. 489–493.
[59] Y. Lierler, M. Maratea, F. Ricca, Systems, engineering environments, and competitions, AI Mag. 37 (3) (2016) 45–52, http://www.aaai.org/ojs/index.php/aimagazine/article/view/2675.
[60] V. Lifschitz, H. Turner, Splitting a logic program, in: P.V. Hentenryck (Ed.), Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, June 13-18, 1994, MIT Press, 1994, pp. 23–37.
[61] M.J. Maher, Constrained dependencies, Theor. Comput. Sci. 173 (1) (1997) 113–149, https://doi.org/10.1016/S0304-3975(96)00193-4.
[62] D. Maier, The Theory of Relational Databases, Computer Science Press, 1983, http://web.cecs.pdx.edu/%7Emaier/TheoryBook/TRD.html.
[63] M. Ostrowski, T. Schaub, ASP modulo CSP: the clingcon system, Theory Pract. Log. Program. 12 (4–5) (2012) 485–503, https://doi.org/10.1017/S1471068412000142.
[64] C. Papadimitriou, M. Yannakakis, A note on succinct representations of graphs, Inf. Comput. 71 (1985) 181–185.
[65] A. Polleres, From SPARQL to rules (and back), in: C.L. Williamson, M.E. Zurko, P.F. Patel-Schneider, P.J. Shenoy (Eds.), Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007, ACM, 2007, pp. 787–796.
[66] C. Redl, The DLVHEX system for knowledge representation: recent advances (system description), CoRR abs/1607.08864, http://arxiv.org/abs/1607.08864, 2016.
[67] F. Ricca, The DLV Java wrapper, in: F. Buccafurri (Ed.), Joint Conference on Declarative Programming, AGP-2003, Reggio Calabria, Italy, September 3-5, 2003, 2003, pp. 263–274.
[68] F. Ricca, L. Gallucci, R. Schindlauer, T. Dell'Armi, G. Grasso, N. Leone, OntoDLV: an ASP-based system for enterprise ontologies, J. Log. Comput. 19 (4) (2009) 643–670, https://doi.org/10.1093/logcom/exn042.

[69] M. Samer, Variable dependencies of quantified CSPs, in: I. Cervesato, H. Veith, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning, Proceedings of 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008, in: Lecture Notes in Computer Science, vol. 5330, Springer, 2008, pp. 512–527.
[70] M. Samer, S. Szeider, Backdoor sets of quantified boolean formulas, J. Autom. Reason. 42 (1) (2009) 77–97, https://doi.org/10.1007/s10817-008-9114-5.
[71] T. Schaub, S. Woltran, Answer set programming unleashed!, Künstl. Intell. 32 (2–3) (2018) 105–108, https://doi.org/10.1007/s13218-018-0550-z, editorial, Special Issue on Answer Set Programming.
[72] P. Schüller, The hexlite solver - lightweight and efficient evaluation of HEX programs, in: F. Calimeri, N. Leone, M. Manna (Eds.), Logics in Artificial Intelligence - Proceedings of 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, in: Lecture Notes in Computer Science, vol. 11468, Springer, 2019, pp. 593–607.
[73] P. Schüller, A. Weinzierl, Answer set application programming: a case study on Tetris, in: M.D. Vos, T. Eiter, Y. Lierler, F. Toni (Eds.), Proceedings of the Technical Communications of the 31st International Conference on Logic Programming, ICLP 2015, Cork, Ireland, August 31 - September 4, 2015, in: CEUR Workshop Proceedings, vol. 1433, CEUR-WS.org, 2015, http://ceur-ws.org/Vol-1433/tc_17.pdf.
[74] D.E. Strode, A dependency taxonomy for agile software development projects, Inf. Syst. Front. 18 (1) (2016) 23–46, https://doi.org/10.1007/s10796-015-9574-1.
[75] B. Susman, Y. Lierler, SMT-based constraint answer set solver EZSMT (system description), in: M. Carro, A. King, N. Saeedloei, M.D. Vos (Eds.), Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA, in: OASICS, vol. 52, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 1:1–1:15.
[76] T. Syrjänen, Including diagnostic information in configuration models, in: J.W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagiv, P.J. Stuckey (Eds.), Computational Logic - CL 2000, First International Conference, London, UK, 24–28 July, 2000, Proceedings, in: Lecture Notes in Computer Science, vol. 1861, Springer, 2000, pp. 837–851.
[77] G. Terracina, N. Leone, V. Lio, C. Panetta, Experimenting with recursive queries in database and logic programming systems, Theory Pract. Log. Program. 8 (2) (2008) 129–165, https://doi.org/10.1017/S1471068407003158.
[78] M. Wolf, Program design and analysis, in: M. Wolf (Ed.), Computers as Components, fourth edition, in: The Morgan Kaufmann Series in Computer Architecture and Desi, Morgan Kaufmann, 2017, pp. 221–319, http://www.sciencedirect.com/science/article/pii/B9780128053874000054.