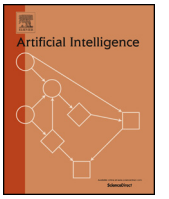
Artificial Intelligence 300 (2021) 103563

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Abstraction for non-ground answer set programs ☆

Zeynep G. Saribatur *, Thomas Eiter, Peter Schüller

Institute of Logic and Computation, TU Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria

ARTICLE INFO

Article history:
Received 20 December 2019
Received in revised form 11 May 2021
Accepted 21 July 2021
Available online 28 July 2021

Keywords:
Abstraction
Answer set programming
Declarative problem solving
Knowledge representation and reasoning
Nonmonotonic formalisms
Explaining unsatisfiability
Counterexample-guided abstraction and refinement

ABSTRACT

Abstraction is an important technique utilized by humans in model building and problem solving, in order to figure out key elements and relevant details of a world of interest. This naturally has led to investigations of using abstraction in AI and Computer Science to simplify problems, especially in the design of intelligent agents and automated problem solving. By omitting details, scenarios are reduced to ones that are easier to deal with and to understand, where further details are added back only when they matter. Despite the fact that abstraction is a powerful technique, it has not been considered much in the context of nonmonotonic knowledge representation and reasoning, and specifically not in Answer Set Programming (ASP), apart from some related simplification methods. In this work, we introduce a notion for abstracting from the domain of an ASP program such that the domain size shrinks while the set of answer sets (i.e., models) of the program is over-approximated. To achieve the latter, the program is transformed into an abstract program over the abstract domain while preserving the structure of the rules. We show in elaboration how this can be also achieved for single or multiple sub-domains (sorts) of a domain, and in case of structured domains like grid environments in which structure should be preserved. Furthermore, we introduce an abstraction-&-refinement methodology that makes it possible to start with an initial abstraction and to achieve automatically an abstraction with an associated abstract answer set that matches an answer set of the original program, provided that the program is satisfiable. Experiments based on prototypical implementations reveal the potential of the approach for problem analysis, by its ability to focus on the parts of the program that cause unsatisfiability and by achieving concrete abstract answer sets that merely reflect relevant details. This makes domain abstraction an interesting topic of research whose further use in important areas like Explainable AI remains to be explored.

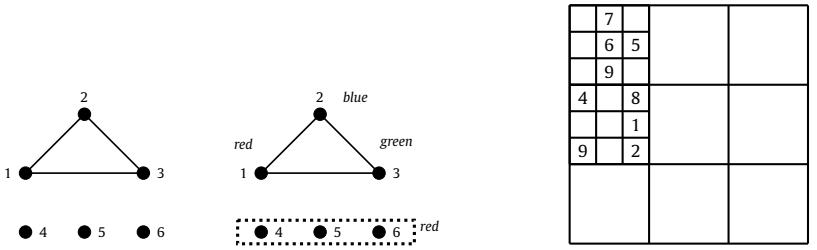
1. Introduction

Abstraction is a technique applied in human reasoning and understanding, by reasoning over the models of the world that are built mentally [30,68]. Although its meaning comes from “to draw away”, there is no precise definition that is capable of covering all meanings that abstraction has in its utilizations. There is a variety of interpretations in different

☆ Some of the results in this article were presented in preliminary form at JELIA 2019 [113] and XAI 2019 [45]. This work has been partially supported by the Austrian Science Fund (FWF) grant W-1255 and by the EU's H2020 research and innovation programme under grant agreements 825619 (AI4EU) and 952026 (HumanE-AI Net).

* Corresponding author.
E-mail addresses: zeynep@kr.tuwien.ac.at (Z.G. Saribatur), eiter@kr.tuwien.ac.at (T. Eiter), ps@kr.tuwien.ac.at (P. Schüller).

https://doi.org/10.1016/j.artint.2021.103563



(a) 3-coloring of a graph (b) Sudoku

Fig. 1. Use of abstraction.

disciplines such as Philosophy, Cognitive Science, Art, Mathematics and Artificial Intelligence, with the shared consensus of the aim to "distill the essential" [108]. Among them is the capability of abstract thinking, which is achieved by removing irrelevant details and identifying the "essence" of a problem [71]. The notion of relevance is especially important in problem solving, as a problem may become too complex to solve if every detail is taken into account. A good strategy to solve a complex problem is to start with a coarse solution and then refine it by adding back more details. When planning a trip, for instance, one may first pick the destination and determine a coarse travel plan; fleshing out the precise details of the travel, such as taking the subway to the airport, comes later. This may be done in a hierarchy of levels of abstraction, with the lowest level containing all of the details. Another view of abstraction is the generalization aspect, which is singling out the relevant features and properties shared by objects. For example, features of an airplane such as color and cargo capacity with their possible differences may be irrelevant to the travel plan; we are (mostly) only interested in the fact that there is an airplane that takes us from Vienna to New York, say. Overall, the general aim of abstraction is to simplify the problem at hand to one that is easier to understand and deal with.

For solving combinatorial problems and figuring out the key elements, humans arguably employ abstraction. In Artificial Intelligence, such problems vary from planning problems like in which order to move blocks to achieve a final configuration, to solving constraint problems such as finding an admissible coloring of the nodes of a given graph. In the latter problem, for instance, isolated nodes can be viewed as a single node and colored the same without thinking about the specific details (Fig. 1a). If a given graph is non-colorable, then we may try to find some subgraph (e.g., a clique) which causes the unsolvability, and we would not care about other nodes in the graph. Similarly with the blocks: if the labels are not important, we would disregard them when figuring out the actions. If the goal configuration cannot be achieved from the initial one, we would aim to find out the particular blocks that cause this.

Notably, such disregard of detail also occurs for problems with multi-dimensional structures such as grid-cells in the well-known Sudoku problem, where a partially filled 9×9 board must be completed by filling in numbers 1..9 into the empty cells under constraints. If an instance is unsolvable, the reason can only be meaningfully grasped by a human by focusing on the relevant sub-regions, as looking at the whole grid is too complex. For illustration, Fig. 1b shows the sub-regions of an instance that contain the reason why no solution exists: as 6 and 7 occur in the middle column, they must appear in the sub-region below in the left column, which is unfeasible as there is only one empty cell. All these examples demonstrate abstraction abilities of humans that come naturally.

Due to its important role in knowledge representation and in reasoning, abstraction has been explored in AI research early on as a useful tool for problem solving: solve a problem at hand first in an abstracted space, and then use the abstract solution as a heuristic to guide the search for a solution in the original space [70,92,106]. This approach was used in planning for speeding up the solving [64] and especially for computing heuristic functions to guide the plan search in the state space. Several abstraction methods were introduced towards this direction, especially to automatically compute abstractions that give a good heuristic [38,61,116]. However, it is well known that the success in solving a problem relies on how "good" the abstraction is. For this, theoretical approaches for defining abstractions with desired properties have been investigated [59,90]. Apart from gaining efficiency (which however may not always materialize [8,63]), abstraction forms a basis to obtain high-level explanations and an understanding of a problem. For more details on these and other related works see Section 7.3.

Abstraction has been studied in other areas of AI and Computer Science as well, among them model-based diagnosis [23,89], constraint satisfaction [13,52], theorem proving [102], to name a few. Particularly fruitful were applications in model checking, which is a highly successful approach to computer aided verification [27], to tackle the state explosion problem by property preserving abstractions [26,32,82]. Furthermore, the seminal counterexample guided abstraction refinement (CEGAR) method [25] allows for automatic generation of such abstractions, by starting from an initial abstraction that over-approximates the behavior of a system to verify, and then stepwise refining the abstraction as long as needed, i.e., as long as spurious (false) counterexamples exist.

Abstraction for Answer Set Programming. Answer Set Programming (ASP) [18,79] is a declarative problem solving paradigm that is rooted in knowledge representation, logic programming, and nonmonotonic reasoning. A problem is represented by a non-monotonic logic program whose answer sets (also called "stable models" [57]) correspond to the solutions of the problem. Thanks to the availability of efficient solvers and the expressiveness of the formalism, ASP has been gaining

popularity for applications in many areas of AI and beyond, cf. [47–49] and references therein, from combinatorial search problems (e.g. configuration, diagnosis, planning) over system modeling (e.g., behavior of dynamic systems, beliefs and actions of agents) to knowledge-intensive applications (e.g., query answering, explanation generation), to name a few [47]. The declarative nature of ASP enables a flexible use for solving different reasoning problems, and it provides a useful basis for investigating ways to help in understanding a problem with its key elements. Studies in understanding how ASP programs find a solution (or none) to a problem have been conducted, which mainly focus on debugging answer sets [16,55,93] or finding justifications [20,103,115]. These approaches could be used to aid in understanding the problem at hand; however, as noted in [50], the explanations offered may contain a high number of details which prevent one from seeing the crucial parts. This is where abstraction would come in handy and could be very fruitfully used.

Somewhat surprisingly, abstraction has not been considered much in the context of nonmonotonic knowledge representation and reasoning, including ASP as a premier formalism in this area. Simplification methods such as equivalence-based rewriting [53,97], partial evaluation [17,67], or forgetting (see [73] for a recent survey), have been extensively studied; however, they strive for preserving semantics, while abstraction provides an approximation of the answer sets of a logic program, in a modified language. We aim here at an approximation in which no answer set is lost, i.e., each answer set of the original program corresponds to some answer set of the abstract program. This enables us to distill all answer sets of the original program from the abstract answer sets and to perform sound reasoning from the abstract answer sets. Specifically, *spurious* answer sets, i.e., abstract answer sets that do not correspond to some answer set of the original program, may be discarded, while under this over-approximation cautious reasoning from all abstract answer sets ensures soundness with respect to cautious reasoning from all answer sets of the original program; moreover, in case no spurious answer set exists we also have completeness (in particular, if no abstract answer set exists). This makes abstraction an interesting topic for research.

In a recent work [110,111], a notion of abstraction for ASP was introduced that focuses on omission of atoms from the vocabulary and ensures over-approximation by rewriting the rules of a given program. That approach is propositional in nature and related to forgetting [112], with the difference of over-approximation vs. preserving the answer sets. Furthermore, it is for unsatisfiable programs related to minimal unsatisfiable subset/core (MUS) extraction from SAT instances and propositional answer set programs, which aims at identifying a smallest set of clauses, respectively rules or asserted literals, that prohibit to have a model, respectively answer set, cf. [1,2,5,76,85]. This may be exploited to give an explanation of non-3-colorability of a graph or non-solvability of a Sudoku instance as in Fig. 1 in terms of clauses, respectively ground literals and rules; however, such an explanation lacks structure in terms of showing subgraphs or subareas, say, which must be extracted in post processing from the propositional encoding. For further discussion of omission abstraction and MUS extraction, we refer to [111].

We follow an orthogonal approach to [110,111] and introduce in this work a notion of abstraction for ASP *on the first-order level* that is concerned with collapsing (i.e., clustering) objects in the (Herbrand) domain of a program. It is that in this way, multiplicity is removed in the spirit of Occam's razor.[1] If the graph in Fig. 1a is represented by facts $node(1), \ldots, node(6)$ and $edge(1,2)$, $edge(1,3)$, $edge(2,3)$, then collapsing the nodes 4, 5, and 6 into an abstract node a would not affect 3-colorability of the graph; we thus expect that an abstract version of an ASP program that encodes all 3-colorings of the graph would yield answer sets from which these 3-colorings can be recovered. However, if there were an edge between 4 and 5, then over-approximation would yield that this abstract program will have spurious answer sets, as the abstract node a may have a single color in some abstract answer set I' while 4 and 5 must have in every original answer set I different colors, thus I cannot be mapped to I'. Similarly, if nodes 1 and 2 were collapsed, the graph would become 2-colorable (where 1 and 2 share the same color), while the original graph is not 2-colorable. These simple examples show that a naive use of domain abstraction – just replace individuals by a cluster of elements – does not work; and yet more subtle effects may surface when programs have recursive definitions such as reachability.

In fact, suitable domain abstraction for ASP is non-trivial and has several challenges. First, the abstract program should be automatically constructed, while the structure of the original rules should be preserved if this is feasible. Second, abstraction refinement, i.e., unclustering of objects for eliminating spurious answer sets, should be automated as well. This is non-trivial, given a large space of possible refinements and that objects might be related among each other in multiple ways, e.g. in temporal or spatial relationships as in reasoning about actions, for instance. And third, the capability of dealing with structure and to support hierarchical abstraction that can handle objects of different granularities at different levels of abstraction is needed.

Contributions. We address the issues above in this work, whose main contributions can be summarized as follows.

(1) We formally introduce the notion of domain abstraction for ASP programs. To this end, we define abstraction mappings m from the original (concrete) domain D of the program to an abstract domain D', and construct an abstract program Π' over D' such that each answer set I of Π maps to an abstract answer set I' of Π'. We propose a systematic approach for the construction of such an abstract program, which works modularly on the syntactic level and transforms each rule in Π into a set of abstract rules with a similar structure. In the transformation, built-in relations and in particular equality, whose treatment provides the backbone of the method, are lifted to the abstract level, and uncertainty caused by the abstracted

[1] Often referred to as "Entia non sunt multiplicanda praeter necessitatem." that is, entities should not be multiplied beyond necessity.

domain D' is carefully respected. Our notion of abstraction can be used for different applications such as obtaining abstract solutions from ASP programs or showing reasons of unsatisfiability in case no answer set exists. We illustrate this on various problems expressed in ASP, among them problems from combinatorial search, planning, and agent behavior assessment.

(2) We present a method which, in case an abstract answer set I' of a program Π w.r.t. mapping m is spurious, computes a refinement m' of the abstraction in order to eliminate I'. To this end, we reduce the test for spuriousness to unsatisfiability of a non-ground ASP program Π' constructed from I', Π, and m. As unsatisfiability of Π' as such leaves one clueless about how to modify m in order to eliminate spuriousness, we present a method for catching causes of the unsatisfiability. The method uses a debugging technique via meta-programming for ASP and obtains useful information for computing a promising refinement m' that removes the spuriousness. Building on an ASP solver, it uses under the hood assumption-based search. To this end, we lift the SPOCK approach [16] for tight programs to the non-ground level such that decisions on refinements can be based on special atoms computed during the debugging. Intuitively, these atoms single out changes for a spurious answer set I' towards a corresponding answer set I of the original program Π, where always some such atoms will be found; based on heuristics, we introduce different refinement strategies to eliminate spurious answer sets. These strategies are employed in a CEGAR-style [25] methodology of iterative abstraction and refinement, which starts with a highly coarse abstraction and automatically searches for and outputs an abstraction with a non-spurious (concrete) answer set if one exists.

(3) We introduce the possibility of multi-dimensional abstraction mappings over a domain. While the abstraction method from above can deal with sorts, we have to modify it to form an abstraction over the relations that is akin to existential abstraction [25], which can be viewed as a simplification of the initially presented abstraction method, in order to enable that elements at mixed levels of abstraction can be handled properly. This in fact is needed to relate in Fig. 1b cells like the top-left corner with abstract cells such as the mid-left 3×3 sub-region, and to express their abstract locations such as being above, left-of etc. We extend the abstraction-&-refinement methodology with handling the structural aspects of grid-cells by using an abstraction of quad-trees, a tree structure often used to partition 2-dimensional spaces, and we consider more sophisticated decision making for the refinement to observe its effects on the resulting abstractions.

(4) We analyze semantic and computational properties of the abstraction approach, which can be exploited for modeling and for guiding the design of suitable implementations. Among other results, we establish that abstractions for sequences of refinements can be built incrementally (Proposition 3.6), and that abstractions of independent sorts can be naturally composed (Proposition 3.7). Furthermore, the two variants of domain abstractions we consider are semantically equivalent, which we demonstrate here for the basic case (Theorem 5.1), but have features making them attractive in different contexts. As regards complexity, we show that checking whether an abstract answer set of a normal logic program is spurious is coNEXP-complete and that deciding whether an abstraction mapping is faithful, i.e., has no spurious abstract answer sets, is $\text{coNEXP}^{\text{NP}}$-complete. Thus, the worst case complexity of these problems coincides with the one of unsatisfiability testing of normal respectively disjunctive non-ground ASP programs [33]. Furthermore, if predicate arities are bounded by a constant, the problems are Π_2^p-complete and Π_p^3-complete, respectively, and thus again have the same complexity as unsatisfiability testing of normal and disjunctive non-ground ASP programs, respectively in the bounded predicate case [39]. Reducing spurious checking to unsatisfiability testing of normal logic programs is thus worst-case optimal in both settings.

(5) We have implemented the abstraction-&-refinement approach in prototypical tools DASPAR and mDASPAR for plain and multi-dimensional abstraction, respectively. They take as input a non-ground program Π and an initial coarse domain mapping m (which by default is the trivial mapping that clusters all elements), and output a refinement mapping m' and an abstract answer set I' for the abstract program Π' that is non-spurious, if one exists; otherwise, i.e., in case Π is unsatisfiable, they provide a refinement mapping m' that is faithful, i.e., such that Π' has no abstract answer sets. The implementations include different refinement strategies and support independent sorts as well as 2-dimensional abstractions with a quad-tree-style refinement process. Based on these tools, we have conducted an experimental evaluation of the approach, where one set of experiments focused on finding non-trivial abstractions for problems expressed by well-known ASP programs (graph coloring, scheduling), while another one consisted in detecting the unsolvability of several benchmark problems involving grid-cells. In the experiments, different debugging strategies were considered and a measure for assessing the quality of multi-dimensional abstraction was developed. The results show the potential of the approach, where in particular a small user-study for a natural grid-cell problem indicates its capability of putting a human-like focus on the grid to show unsatisfiability.

Summarizing, our work on domain abstraction for ASP opens an intriguing line of research which aims at making it possible to identify the gist of a program's domain that is responsible for matters such as inconsistency or certain solutions of interest. The approach that we present provides the ability to adjust the granularity of abstraction towards the details relevant for a problem in an (semi-)automated way. The experimental results indicate the value of domain abstraction for program analysis, whose further use in important areas like Explainable AI remains to be explored.

Organization. The remainder of this article is organized as follows. The next section recalls notions from ASP as needed for this work, and reviews the seminal approach to abstraction over (in essence) propositional ASP programs introduced in [110,111]. After that, in Section 3, we turn to domain abstraction for non-ground ASP programs, where we present an abstraction procedure, consider various extensions, and study semantic and computational properties of the approach. In the subsequent Section 4, we present our refinement methodology that is based on debugging of ASP programs. After Sections 3

and 4 we are well equipped to apply the CEGAR-style abstraction and refinement procedure for answer set programs. As a further consideration, in Section 5, we focus on multi-dimensional abstraction and an alternative abstraction method, based on existential abstraction, that is needed for it, which can also be combined with the refinement method from Section 4 to apply the CEGAR-style procedure. Implementation and evaluation of the approach for both abstraction types is considered in Section 6, while in Section 7 we discuss further notions of abstraction and possible use cases, as well as related work. The final Section 8 provides a summary and outlines issues for future research.

In order not to distract from the flow of reading, longer proofs and further details have been moved to the Appendix (A and B), which also provides a further use case in agent behavior assessment (C).

2. Background

2.1. Answer set programming

In this section, we recall the concepts and notions of Answer Set Programming (ASP) that we need for this article. We refer to [18,19,114] for more background and references. We start with syntax and semantics of ASP programs and then recall some notions that are useful for this work.

Syntax. We consider a first-order vocabulary $\mathcal{V} = (\mathcal{P}, \mathcal{C})$ consisting of non-empty finite sets $\mathcal{P}$ of predicates and $\mathcal{C}$ of constants. Let $\mathcal{X}$ represent the set of variable symbols. A *term* is either a constant from $\mathcal{C}$ or a variable from $\mathcal{X}$. An *atom* is an expression α of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ and each t_i is a term; $n \geq 0$ is the *arity* of p, and $arg(\alpha) = \{t_1, \ldots, t_n\}$ denotes the set of arguments of α. Atoms are called *propositional* if $n = 0$ and *ground* if they do not contain variables. A *literal* is either a formula α (*positive literal*) or *not* α (*negative literal*), where α is an atom. Intuitively, a negative literal *not* α is true if α cannot be derived using rules, and false otherwise; *not* is called *weak* or *default* negation.

Definition 2.1. A *normal logic program* Π is a finite set of *rules*. A *rule* r is an expression of the form

$$\alpha_0 \leftarrow \alpha_1, \ldots, \alpha_m, not\ \alpha_{m+1}, \ldots, not\ \alpha_n, \quad 0 \leq m \leq n. \tag{1}$$

We refer to α_0 as the *head* of r, and $\alpha_1, \ldots, \alpha_m, not\ \alpha_{m+1}, \ldots, not\ \alpha_n$ as the *body* or r. We also write r as $\alpha_0 \leftarrow B(r)$, such that $H(r) = \alpha_0$ denotes the head or as $H(r) \leftarrow B^+(r), not\ B^-(r)$, where $B^+(r) = \{\alpha_1, \ldots, \alpha_m\}$ is the *positive body* and $B^-(r) = \{\alpha_{m+1}, \ldots, \alpha_n\}$ is the *negative body* of r. We may omit r from $B(r)$, $B^+(r)$ etc. if r is clear. We refer to "normal logic programs" as "programs" unless otherwise noted.

A rule r is a *constraint* if α_0 is the propositional atom $\bot$ (then also omitted), where $\bot$ is a predicate constant for falsity. Furthermore, r is a *fact* if $n = 0$ and no variable occurs in r, and r is *positive* if $n = m$. A rule is *ground* if all literals occurring in it are ground.

A program Π is *positive* and *ground* if the rules in it have the respective property; it is *safe* if every variable that occurs in a rule also occurs in the same rule in some positive body literal. We assume safety of the programs which usually is achieved by making use of *domain predicates*, which are unary predicates that are given as facts and are true for all constants that the variables in the program are being grounded over. We use predicate *dom* for the domain predicate. For simplicity of presentation, we omit *dom* in the representation, unless it is necessary to ensure safety, and from the answer sets.

Choice rules are syntactic sugar of the form $\{a\} \leftarrow B$, which stands for the rules $a \leftarrow B, not\ \overline{a}$ and $\overline{a} \leftarrow B, not\ a$, where $\overline{a}$ is a new atom.

Semantics. The answer set semantics is defined via ground programs. For a program Π, we define its *ground instantiation* as follows.

Given a program Π, its *Herbrand universe*, denoted by HU_Π, is the set of all constant symbols $C \subseteq \mathcal{C}$ appearing in Π; in case there is no constant symbol, then $HU_\Pi = \{c\}$ for some arbitrary constant symbol. The *Herbrand base* of a program Π, denoted by HB_Π, is the set of all ground atoms constructed using predicates from $\mathcal{P}$ and constants from $\mathcal{C}$.

The *ground instances of a rule* $r \in \Pi$, denoted by $grd(r)$, is the set of rules obtained by replacing all variables in r with constant symbols in HU_Π in all possible ways. The *grounding* of a program Π is then $grd(\Pi) = \bigcup_{r \in \Pi} grd(r)$. To group the rules in $grd(\Pi)$ with the same head α, we use $def(\alpha, \Pi) = \{r \in grd(\Pi) \mid H(r) = \alpha\}$, to group non-ground rules in Π that define atoms with predicate p, we use $pdef(p, \Pi) = \{r \in \Pi \mid H(r) = p(t_1, \ldots, t_n)\}$.

Let Π be a ground program. An *interpretation* I is a subset of HB_Π and I *satisfies* a rule $r \in \Pi$, denoted by $I \models r$, if $H(r) \in I$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. An interpretation is a *model* of Π, denoted by $I \models \Pi$, if $I \models r$ for all $r \in \Pi$. A model I is *minimal* if there is no model J of Π such that $J \subset I$.

Example 2.1. Consider the program Π below and the interpretation $I = \{a, b, d\}$.

$c \leftarrow not\ d.$

$d \leftarrow not\ c.$

$a \leftarrow not\ b, c.$

$b \leftarrow d.$

I is a model of Π, but it is not minimal, since the interpretation $I' = \{b, d\}$ is also a model of Π.

Definition 2.2 *(GL-reduct).* The *Gelfond-Lifschitz* (GL-)*reduct* of a program Π relative to an interpretation $I \subseteq HB_\Pi$, denoted by Π^I, is the ground positive program obtained from $grd(\Pi)$ when each rule $H(r) \leftarrow B^+(r), not\ B^-(r)$

(i) with $B^-(r) \cap I \neq \emptyset$ is deleted, and
(ii) is replaced by $H(r) \leftarrow B^+(r)$, otherwise.

Informally, the first step is to remove the rules where I contradicts a default negated literal, and from the remaining rules, the second step removes their negative body.

Definition 2.3. An interpretation I is an *answer set* of a program Π if it is the minimal model of the GL-reduct Π^I.

Apart from the GL-reduct which is considered to be the standard definition for stable models (i.e., answer sets), a collection of other definitions can be found in [78].

The set of answer sets of a program Π is denoted as $AS(\Pi)$. A program Π is *unsatisfiable* if $AS(\Pi) = \emptyset$.

Example 2.2 *(ctd).* Π has two answer sets, viz. $I_1 = \{c, a\}$ and $I_2 = \{d, b\}$; indeed,

- $\Pi^{I_1} = \{c.;\ a \leftarrow c.; b \leftarrow d.\}$ and I_1 is the minimal model of Π^{I_1}; similarly,
- $\Pi^{I_2} = \{d.;\ b \leftarrow d.\}$ has I_2 as its minimal model.

Dependencies. The *dependency graph* of a ground program Π is a directed graph $G_\Pi = (V, E)$, where the vertices V equal HB_Π, and the edges $E = E^+ \cup E^-$ consist of positive edges E^+ from any $q = H(r)$ to any $p_1 \in B^+(r)$ and negative edges E^- from any $q = H(r)$ to any $p_2 \in B^-(r)$, for all $r \in \Pi$.

Example 2.3 *(ctd).* G_Π has positive edges $a \to c$ and $b \to d$ and negative edges $c \to d$, $d \to c$ and $a \to b$.

The *positive dependency graph* is the dependency graph containing only the positive edges, denoted by G_Π^+. A program Π is *tight* if G_Π^+ is acyclic. A non-empty set L of ground atoms describes a *positive loop* of Π if for each pair $p, q \in L$ there is a path τ from p to q in G_Π^+ such that each atom in τ is in L.

As we consider non-ground programs, we need to take care of cyclic dependencies of atoms at the non-ground level. For that, for a given non-ground Π, we consider a *non-ground dependency graph* $G_\Pi^{ng} = (V, E)$ where the vertices V are the atoms appearing in Π, and the edges $E = E^+ \cup E^-$ consist of positive edges E^+ from any $a_1(\overline{x}_1) = H(r)$ to any $a_2(\overline{x}_2) \in B^+(r)$ and negative edges E^- from any $a_1(\overline{x}_1) = H(r)$ to any $a_2(\overline{x}_2) \in B^-(r)$, for all $r \in \Pi$. A *non-ground negative dependency cycle* of length $n \geq 2$ is of the form

$$a_1(\overline{x}_1) \to a_2(\overline{x}_2) \to \ldots \to a_n(\overline{x}_n) \to a_1(\overline{x}_1)$$

where $\alpha \to \alpha'$ denotes that there is a path τ in G_Π^{ng} from α to some atom α'' that unifies with α' including only one negative edge. For example, a choice rule consists of a non-ground negative dependency cycle of length 2. In this article, we focus on the predicates a_i of the atoms to determine the dependency, and we thus consider negative dependency cycles of form $a_1 \to a_2 \to \ldots \to a_n \to a_1$. Accordingly, we will consider a set L_c of atoms as cyclic if for each pair $\alpha_1, \alpha_2 \in L_c$ a chain $pred(\alpha_1) \to \ldots \to pred(\alpha_2)$ exists. Two cyclic sets L_{c_1}, L_{c_2} of atoms are called *independent* if they do not share an atom, i.e., $L_{c_1} \cap L_{c_2} = \emptyset$.

ASP solvers first generate a grounding of the given program, and then the search for an answer set is conducted over the ground program.

2.2. Abstraction in ASP

Abstraction aims at discarding some details of a problem to obtain a more high-level view of a solution. This can be done in an over-approximation, which means that each original solution has some corresponding solution in the abstraction but one may encounter abstract solutions which do not have a corresponding original solution.

In [110], the authors introduced such a notion of abstraction in ASP for over-approximating a given ground program Π on vocabulary $\mathcal{A}$, by constructing a ground program Π' on a smaller vocabulary $\mathcal{A}'$, i.e., $|\mathcal{A}| \geq |\mathcal{A}'|$, More formally, abstraction was defined at the semantic level as follows.

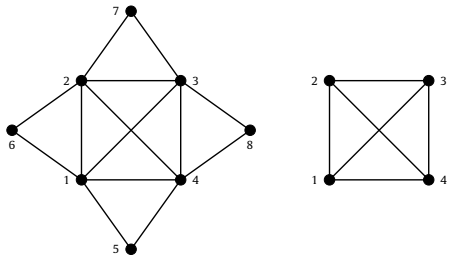


Fig. 2. Non-3-colorable graph.

Definition 2.4 *(cf. [110]).* Given two programs Π and Π' with $|\mathcal{A}| \geq |\mathcal{A}'|$, where $\mathcal{A}, \mathcal{A}'$ are sets of ground atoms of Π and Π', respectively, and a mapping $m : \mathcal{A} \to \mathcal{A}' \cup \{\top\}$, where $\top$ is a propositional constant for truth, we call Π' an *abstraction* of Π with respect to m, if for every answer set I of Π, $I' = \{m(\alpha) \mid \alpha \in I\}$ is an answer set of Π'. Furthermore, Π' is an *abstraction* of Π, if some such mapping m exists.

We refer to m as an *abstraction mapping* which often will be tacitly assumed to be given. This notion of abstraction gives us the possibility to do clustering over atoms of the program. The (abstract) program Π' on the smaller vocabulary $\mathcal{A}'$ serves to represent abstract answer sets. While the reduced vocabulary eases the search for an (abstract) answer set I', an additional check is needed whether the original program Π has some answer that maps to I'. In [110], the focus was on abstraction by omitting atoms from a program, i.e., by clustering them into $\top$.

Example 2.4. Consider the program that describes the graph 3-coloring problem below (adapted from the coloring encoding in the ASP Competition 2013) and the graphs shown in Figs. 1a and 2.

$$
\begin{aligned}
color(red).\ color(green).\ color(blue).& \\
\{chosenColor(N, C)\} \leftarrow node(N), color(C).& \\
colored(N) \leftarrow chosenColor(N, C).& \\
\bot \leftarrow not\ colored(N), node(N).& \\
\bot \leftarrow chosenColor(N, C_1), chosenColor(N, C_2), C_1 \neq C_2.& \\
\bot \leftarrow chosenColor(N_1, C), chosenColor(N_2, C), edge(N_1, N_2).&
\end{aligned}
\tag{2}
$$

If we omit in the (ground version) of the encoding (2) with the instance shown in Fig. 1a all atoms involving the nodes $4, 5, 6$, the resulting abstract program will have answer sets which all correspond to some answer set of the original program, as the omitted nodes can be colored arbitrarily without destroying 3-colorability.

We could likewise map all atoms α involving $4, 5, 6$ to atoms α' in which these nodes are replaced by a new node k; e.g., $node(4)$ would become $node(k)$, $node(5)$ becomes $node(k)$ etc. The abstract answer sets correspond then again to original answer sets, as the coloring of $4, 5, 6$ does not matter. On the other hand, if we consider the graph in Fig. 2 and omit all atoms that involve nodes $5, 6, 7, 8$, then the resulting abstract program has no answer sets, as the remaining clique $1 - 2 - 3 - 4$ is not 3-colorable; also the original program has no answer set.

The latter observations are not by accident but in fact a useful property.

Proposition 2.1. *Let Π' be an abstraction of Π. If $AS(\Pi') = \emptyset$, then we have $AS(\Pi) = \emptyset$.*

In general, over-approximation can cause abstract answer sets that have no corresponding original answer set.

Definition 2.5 *(spurious & concrete answer sets).* Let Π' be an abstraction of Π for the mapping m. The answer set $I' \in AS(\Pi')$ is *concrete* if there exists an answer set $I \in AS(\Pi)$ such that $m(I) = I'$; otherwise, it is *spurious*.

In these terms, the abstract answer sets of the first abstract program constructed in Example 2.4 are all concrete, while if we drop all atoms involving nodes $1, 2, 3, 4$ from the graph in Fig. 2, the abstract answer sets of the resulting abstract program are all spurious.

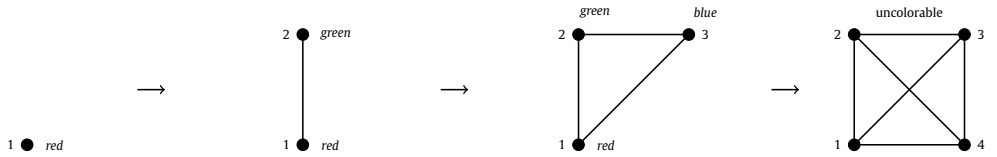


Fig. 3. Abstraction refinement upon spurious graph colorings.

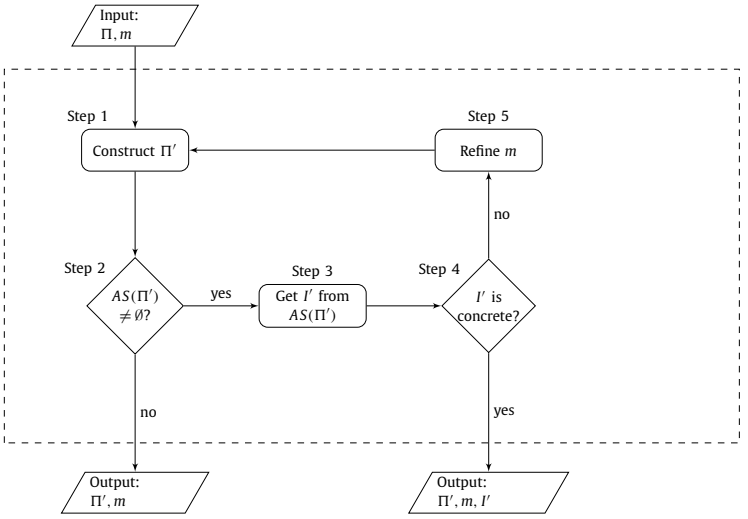


Fig. 4. Abstraction & Refinement Methodology.

2.2.1. Abstraction refinement methodology

To get rid of spurious abstract answer sets, the abstraction mapping m needs to be *refined* to a more fine-grained abstraction; in case of omission abstraction, the refinement would be to add back some of the omitted atoms.

We consider a CEGAR-style [25] abstraction refinement approach which refines an initial abstraction repeatedly until a concrete solution is found or inconsistency (i.e., absence of solutions) is detected.

Before describing the general methodology, we first illustrate the idea with the graph coloring example.

Example 2.5 *(ctd).* Fig. 3 shows the abstraction of omitting 7 of the 8 nodes and their edges. Deciding on a color for the remaining node is easy. However, in the original domain, no coloring can match node 1 colored *red* as the graph is uncolorable. A refinement of this abstraction would be to add back some of the nodes and the knowledge about the edges. Until an abstraction is achieved where the four nodes causing the uncolorability are distinguished, a spurious coloring always occurs.

Fig. 4 depicts the methodology introduced in [110]. For a program Π, we start with an initial abstraction mapping m to construct an abstract program Π' (Step 1) that over-approximates the original program Π and then compute the abstract answer sets. Over-approximation guarantees that if Π has an answer set I, then a corresponding abstract answer set $m(I)$ of the abstract program Π' exists. If in turn the abstract program Π' has no answer set (Step 2), by Proposition 2.1 Π is unsatisfiable. In this case, the abstract program Π' and the mapping m are returned. When we pick an abstract answer set $I' \in AS(\Pi')$ (Step 3), we check for concreteness (Step 4). If I' is concrete, it shows a solution to Π; in this case, the abstract program Π', the mapping m and the concrete abstract answer set I' are returned. If I' is spurious, we refine the abstraction mapping m to m' (Step 5) and loop back to Step 1. This loop continues until either a picked abstract answer set is concrete, or the abstract program has no answer sets. Termination is guaranteed as in the extreme case m is refined to the trivial identity mapping, i.e., each element of the original domain is mapped to itself; Π' will coincide with Π. Thus, if Π is unsatisfiable, the procedure will stop at Step 2, otherwise at Step 4.

In the next section, we introduce the new abstraction method to be used in Step 1. We investigate an abstraction over non-ground ASP programs given a mapping over their domain (i.e., the Herbrand universe) that singles out the domain elements. The correctness checking of an abstract answer set (Step 4) and then deciding on a refinement (Step 5) is done using a debugging approach which is introduced in Section 4.

3. Domain abstraction

The omission-based abstraction approach in [110,111] is propositional in nature and does not account for the fact that in ASP, non-ground rules talk about a domain of discourse, where for the (non)existence of an answer set, the precise set of elements may not matter, but rather how certain elements are related. For example, the graph coloring encoding (2) expresses that each node should be colored differently from its neighbors. The names of the neighbor nodes are not relevant to the color determination, rather the relation of having a neighbor with a certain chosen color.

In this section, we tackle the issue of automatically constructing and evaluating a suitable abstract program Π' for a given non-ground ASP program Π with an abstraction over its domain.

To illustrate the abstraction and its various challenges, we use the following example.

Example 3.1 *(running example).* Consider the following example program Π with domain $D = \{1, \ldots, 5\}$:

$$a(1).\ a(3).\ c(2).\ d(5). \tag{3}$$

$$b(X, Y) \leftarrow a(X), d(Y). \tag{4}$$

$$e(X) \leftarrow c(X), a(Y), X \neq Y. \tag{5}$$

$$\bot \leftarrow b(X, Y), e(X). \tag{6}$$

Note that Π has the answer set $I = \{a(1), a(3), d(5), c(2), b(1, 5), b(3, 5), e(2)\}$.

The abstraction mapping is defined over the Herbrand universe of Π, called *domain*, by merging the constants.

Definition 3.1. Given a domain D of Π, a *(domain abstraction) mapping* is a surjective function $m : D \to \widehat{D}$ for a set $\widehat{D}$ (the *abstracted domain*).

Thus, a domain abstraction mapping divides D into *clusters* $\{d' \in D \mid m(d') = \hat{d}\}$ of elements that are seen as equal, where $\hat{d} \in \widehat{D}$. If unambiguous, we also write $\hat{d}$ for its cluster $m^{-1}(\hat{d})$.

Example 3.2 *(ctd).* The Herbrand universe for the program Π is $HU_\Pi = \{1, 2, 3, 4, 5\}$. A possible mapping for Π with $\widehat{D}_1 = \{k_1, k_2, k_3\}$ clusters $2, 3$ to the element k_2, $4, 5$ to the element k_3 and 1 to singleton cluster k_1, i.e., $m_1 = \{\{1\} \mapsto k_1, \{2, 3\} \mapsto k_2, \{4, 5\} \mapsto k_3\}$.[2] A naive mapping is $m_2 = \{\{1, \ldots, 5\} \mapsto k\}$ with $\widehat{D}_2 = \{k\}$.

Abstracting the elements in the Herbrand universe induces an abstraction of the Herbrand base. Each domain abstraction mapping m naturally extends to ground atoms $\alpha = p(v_1, \ldots, v_n)$ by

$$m(\alpha) = p(m(v_1), \ldots, m(v_n));$$

we say that α is *mapped to a singleton cluster* if $|m^{-1}(m(\alpha))| = 1$, and is *mapped to a non-singleton cluster* otherwise.

Example 3.3 *(ctd).* In the Herbrand base HB_Π the atoms with the predicates a, b, c, d, e will also be modified according to m, i.e., $a(1), b(1, 5), e(2)$ get changed to $a(k_1), b(k_1, k_3), e(k_2)$; $a(1)$ is mapped to a singleton cluster while $b(1, 5), e(2)$ are mapped to non-singleton clusters.

Given a program Π and an induced mapping $m : \mathcal{A} \to \hat{\mathcal{A}}$ from the Herbrand base $\mathcal{A}$ of Π to $\hat{\mathcal{A}} = m(\mathcal{A}) = \{m(\alpha) \mid \alpha \in \mathcal{A}\}$, we want an abstract program Π' that achieves over-approximation as in Definition 2.4. However, even in the ground case, simply applying m to Π does not work in general. Moreover, we want domain abstraction that for non-ground Π results in a non-ground program Π'. Building a suitable Π' turns out to be challenging and needs to solve several issues, which we discuss in the next section.

3.1. Towards an over-approximation

Given a mapping m that describes an abstraction over the domain of a program Π, we start with the intuition of applying m to each rule, i.e., each atom in a rule is modified according to m, in order to obtain an abstraction Π' of Π.

To this end, we extend the mapping m to the non-ground case as follows.

[2] By abuse of notation, we write $\{e_1, \ldots, e_n\} \mapsto \hat{d}$ for $e_1 \mapsto \hat{d}, \ldots, e_n \mapsto \hat{d}$, where $\{e_1, \ldots, e_n\} = m^{-1}(\hat{d})$.

Definition 3.2 *(extended non-ground mapping).* For any domain mapping $m\colon D \to \widehat{D}$, we let $m(X) = X$ for each $X \in \mathcal{X}$ (i.e., variables remain unchanged). Furthermore, we extend m to atoms $\alpha = p(t_1, \ldots, t_n)$, negated atoms $not\,\alpha$, and sets L of literals by $m(p(t_1, \ldots, t_n)) = p(m(t_1), \ldots, m(t_n))$, $m(not\,\alpha) = not\,m(\alpha)$, and $m(L) = \{m(\lambda) \mid \lambda \in L\}$, respectively.

One may think that simply keeping the rules in Π and considering the abstract domain for evaluation should be enough to achieve an over-approximation. However, this does not hold in general.

Example 3.4 *(ctd).* If we keep the (non-fact) rules (4)-(6) in the program Π the same, but lift the domain and the facts to $\widehat{D}_1$ with the mapping m_1, the resulting program becomes unsatisfiable. Indeed, lifting the facts yields $a(k_1)$, $a(k_2)$, $c(k_2)$, and $d(k_3)$. Thus rule (4) derives $b(k_1, k_3)$ and $b(k_2, k_3)$, while rule (5) derives $e(k_2)$. Consequently, the constraint (6) fires for $X = k_2$ and $Y = k_3$. If we lift the domain to $\widehat{D}_3 = \{k_1, k_2\}$ with the mapping $m_3 = \{\{1, 2, 3\} \mapsto k_1,\ \{4, 5\} \mapsto k_2\}$, then the resulting program has the answer set $\{a(k_1), d(k_2), c(k_1), b(k_1, k_2)\}$ as lifting the facts yields $a(k_1)$, $c(k_1)$, and $d(k_2)$, from which rule (4) derives $b(k_1, k_2)$ while rules (5) does not fire. The original answer set I is not mapped to this abstract answer set as $m_3(e(2)) = e(k_1)$ does not occur in it.

We will now look into why the naive approach may fail to achieve over-approximation and present the idea of our approach to deal with the arising issues.

If we consider the rule (4), then keeping this rule unchanged in the abstraction does not cause issues: it ensures that $b(X, Y)$ holds true in the answer set whenever $a(X)$ and $d(Y)$ hold true, where X and Y can take any value. This is similar for rules in which the arguments of body literals are unconstrained, i.e., they can take independently of each other any value from the domain and no (in)equality relations occur in the body. Issues arise if this is not the case.

Shared arguments. Arguments in a rule body that are shared may cause an issue depending on the abstraction mapping. This is because if an atom $p(k)$ in the abstract program holds true for a cluster k, then this represents that for some, but not necessarily all, original domain elements x in the cluster k the atom $p(x)$ holds true. If another atom $q(k)$ occurs in the body, then both $p(k)$ and $q(k)$ might be true while for some x, x' in k the atoms $p(x)$ and $q(x')$ might have different truth values, and the original rule would not fire. Hence the interaction of shared arguments at the abstract level must be treated by considering different cases of how the involved predicates might behave at the original level.

Instead of doing this ad hoc, we adopt a uniform approach in which the treatment is relegated to auxiliary atoms. To this end, we are *standardizing apart* the shared arguments, which effects that non-relation literals can remain untouched.

Example 3.5 *(ctd).* The constraint (6), i.e., $\bot \leftarrow b(X, Y), e(X)$, has a shared use of the variable X. If the constraint is kept the same, then for the mapping m_1 the rules (4) and (5) derive $b(k_2, k_3)$ and $e(k_2)$ from $a(k_2), d(k_3)$ and $c(k_2), a(k_1)$, respectively. As (6) excludes that both $b(k_2, k_3)$ and $e(k_2)$ are true, no abstract answer set exists. This issue arises as $k_2 = \{2, 3\}$ is a non-singleton cluster and for $X = k_2$, $Y = k_3$ not every atoms $b(x, y)$ and $e(x)$ that are mapped to $b(k_2, k_3)$ and $e(k_2)$, respectively, hold both true in the original answer set, e.g., $b(3, 5), e(2)$. We address this by standardizing apart the multiple occurrence of X, i.e., we replace its second occurrence by a fresh variable X_1 that is equated to X, leading to the rule

$$\bot \leftarrow b(X, Y), e(X_1), X = X_1. \tag{7}$$

In this way, the focus of abstraction is directed towards the relation, i.e., $X = X_1$, in the rule.

In a similar way, also constants that may interact with other arguments in rule bodies are standardized apart. Details are given in Section 3.1.1.

(In)equality relation over the abstract domain. The domain clustering might cause that (in)equality relations in the abstract domain fail to hold in the original domain, resulting in an *uncertainty* to be treated in the abstraction process.

Example 3.6 *(ctd).* The uncertainty caused by applying relation $=$ on the abstract domain is observed in Example 3.5, where although $k_2 = k_2$ holds in the abstraction, not all elements mapped to k_2 satisfy this relation, e.g., $2 \neq 3$.

Consider now the rule (5) and the domain $\widehat{D}_3$ with the mapping m_3 from Example 3.4. There we saw that simply keeping the rules results in an abstract answer set which does not contain $e(k_1)$. This happens since having $a(k_1)$ and $c(k_1)$ makes rule (5) inapplicable as $k_1 \neq k_1$ does not hold true. However, there are elements mapped to k_1 for which relation $\neq$ holds in the original domain, e.g., $2 \neq 3$.

Table 1 shows all the cases of the behavior of the relations in the abstract domain for particular mappings.

In Section 3.1.2, we handle such uncertainties caused by domain clusters by assigning *types* to the relations with respect to the mappings. For example, we will refer to the case where $X = X_1$ holds true in the abstract program while not all elements mapped to X, X_1 satisfy $=$ as type III. The case where $X \neq Y$ is false while in the original domain some elements mapped to X, Y satisfy $\neq$ will be referred to as type IV. For these cases, we turn the head of a rule into a choice in order to capture the possible firing and the non-firing of the original rule in alternative abstract answer sets.

Table 1
Behavior of the relations $=, \neq$ in $\widehat{D}_1$ with the mapping m_1, and in $\widehat{D}_3$ with the mapping m_3, assuming symmetry.

	$=$			$\neq$		
$\widehat{D}_1$	$k_1 = k_1$	$k_2 = k_2$	$k_3 = k_3$	$k_1 \neq k_2$	$k_1 \neq k_3$	$k_2 \neq k_3$
D	$1 = 1$	$2 = 2$ $3 = 3$ $2 \neq 3$	$4 = 4$ $5 = 5$ $4 \neq 5$	$1 \neq 2$ $1 \neq 3$	$1 \neq 4$ $1 \neq 5$	$2 \neq 4$ $2 \neq 5$ $3 \neq 4$ $3 \neq 5$
$\widehat{D}_3$	$k_1 = k_1$	$k_2 = k_2$		$k_1 \neq k_2$		
D	$1 = 1$ $2 = 2$ $3 = 3$ $1 \neq 2$ $1 \neq 3$ $2 \neq 3$	$4 = 4$ $5 = 5$ $4 \neq 5$		$1 \neq 4$ $1 \neq 5$ $2 \neq 4$ $2 \neq 5$ $3 \neq 4$ $3 \neq 5$		

Default negation. For non-singleton clusters, negative literals may also result in losing an original answer set. This is because default negation has issues with uncertainty, similar to (in)equality relations. However, an important difference is that the interpretation of relations depends on the domain, thus they can be evaluated and removed from a program during the instantiation process. So the treatment by looking at the behavior of the relations in the abstract domain for a mapping is capable of handling the uncertainty. This is not possible for defined predicates, since when constructing an abstract program Π' from Π, the original program reduct Π^I, for each answer set I of Π, must be appropriately reflected, such that no answer set I gets lost.

Example 3.7 *(ctd).* Suppose we modify the rule (4) by adding default negation in front of d:

$$b(X, Y) \leftarrow a(X), \mathit{not}\ d(Y), dom(Y). \tag{8}$$

The answer set of the modified program Π is then $I = \{a(1), a(3), d(5), c(2), b(1,1), \ldots, b(1,4), b(3,1), \ldots, b(3,4), e(2)\}$. When the modified rule remains unchanged by abstraction in Π', we obtain with the mapping $m_3 = \{\{1,2,3\} \mapsto k_1, \{4,5\} \mapsto k_2\}$ in the abstract domain $\widehat{D}_3$ the answer set $I' = \{a(k_1), d(k_2), c(k_1), b(k_1,k_1)\}$, which for the atoms $b(1,4)$ and $e(2)$ does not contain their abstractions $m_3(b(1,4)) = b(k_1,k_2)$ and $m_3(e(2)) = e(k_1)$. The reason is that $\mathit{not}\, d(k_2)$ is false in I', and hence the body of the rule (8) is not satisfied. This makes the reduct $\Pi'^{I'}$ contain for (8) only the rule

$$b(k_1, k_1) \leftarrow a(k_1), dom(k_1).$$

and no rule with k_2. However, if we look at the cluster $k_2 = \{4, 5\}$, even though the literal $\mathit{not}\, d(5)$ is false in the original answer set I, still $\mathit{not}\, d(4)$ is true in I. So the original reduct Π^I also has the rules

$$b(1, 4) \leftarrow a(1), dom(4).$$

$$b(3, 4) \leftarrow a(3), dom(4).$$

for (8), which are not represented in $\Pi'^{I'}$.

The example shows that having a negative literal $\mathit{not}\, \alpha$ false, i.e., α true in an abstract interpretation I' does not allow us to conclude that all atoms α' mapped to α are true in an original interpretation I, even if I is an answer set. However, it would be possible if only one atom α' is mapped to α; the latter is ensured if each argument $\hat{d}_i$ of $\alpha = p(\hat{d}_1, \ldots, \hat{d}_n)$ is a singleton cluster.

We deal with this issue by adding a rule for non-singleton clusters, where in the original rule the polarity of $\mathit{not}\, \alpha$ is shifted and the head is turned into a choice. Specifically, for the rule (8) above, the rule

$$\{b(X, Y)\} \leftarrow a(X), d(Y), \mathit{not}\ isSingleton(Y). \tag{9}$$

will be added in the abstract program; here *isSingleton* is an auxiliary predicate that will be introduced in Section 3.2.

In conclusion, we need a fine-grained systematic approach to deal with uncertainties. Furthermore, for Example 3.7 the treatment described above would be sufficient, since the predicate d is not defined in terms of other predicates. One has to pay a particular attention to cycles through negation, which, as well-known in ASP, may create alternative answer sets.

Table 2
Cases for lifting a binary relation $\circ \in \{=, \neq\}$ from D to $\widehat{D}$ according to a mapping $m : D \to \widehat{D}$ and applying the relation to some $\hat{d}_1, \hat{d}_2 \in \widehat{D}$. The symbol $\overline{\circ}$ stands for the complement of the relation $\circ$.

$\tau_{\mathrm{I}}^{\circ}(\hat{d}_1, \hat{d}_2) : \hat{d}_1 \circ \hat{d}_2 \wedge \forall x_1 \in m^{-1}(\hat{d}_1), \forall x_2 \in m^{-1}(\hat{d}_2).\, x_1 \circ x_2$ $\tau_{\mathrm{II}}^{\circ}(\hat{d}_1, \hat{d}_2) : \hat{d}_1 \,\overline{\circ}\, \hat{d}_2 \wedge \forall x_1 \in m^{-1}(\hat{d}_1), \forall x_2 \in m^{-1}(\hat{d}_2).\, x_1 \,\overline{\circ}\, x_2$	certain cases
$\tau_{\mathrm{III}}^{\circ}(\hat{d}_1, \hat{d}_2) : \hat{d}_1 \circ \hat{d}_2 \wedge \exists x_1 \in m^{-1}(\hat{d}_1), \exists x_2 \in m^{-1}(\hat{d}_2).\, x_1 \,\overline{\circ}\, x_2$ $\tau_{\mathrm{IV}}^{\circ}(\hat{d}_1, \hat{d}_2) : \hat{d}_1 \,\overline{\circ}\, \hat{d}_2 \wedge \exists x_1 \in m^{-1}(\hat{d}_1), \exists x_2 \in m^{-1}(\hat{d}_2).\, x_1 \circ x_2$	uncertain cases

3.1.1. Standardizing apart

A rule is *standardized apart* if all non-relational atoms (i.e., those with predicates not in $\{=, \neq, dom\}$) contain only variables and no two non-relational atoms share a variable. In order to allow for a uniform treatment of the interaction of arguments in rule bodies for abstraction, rules are rewritten as follows. Suppose $r : \alpha \leftarrow B(r)$ is a rule of form (1). Then the following steps are performed as long as possible:

1. If a variable X occurs in non-relational atoms $\alpha_i = p_i(t_1^i, \ldots, t_{n_i}^i) \in B^+(r) \cup B^-(r)$, $i = 1, 2$, i.e., $p_1, p_2 \notin \{=, \neq, dom\}$, as t_i^1 and t_j^2, respectively, then replace t_j in α_2 with a fresh variable X' and add $X = X'$ to the rule body; here, $\alpha_1 = \alpha_2$ is allowed but then $i \neq j$ is required, i.e., X occurs in the same atom α at different argument positions i and j.
2. If a constant symbol c occurs in a non-relational atom $\alpha = p(t_1, \ldots, t_n) \in B^+(r) \cup B^-(r)$, where $p \neq dom$, as t_j, then replace t_j in α with a fresh variable X' and add $X' = c$ to the rule body.

If none of the steps 1-2 is applicable, r is *standardized apart*.

Example 3.8 *(ctd).* Applying the above procedure to the rule (6), we obtain the standardized apart form (7). The rule

$$q(X) \leftarrow p(X, X, Z), q(Y, W), not\, r(3, Y, V), dom(V), Z \neq 1, Y \neq W. \tag{10}$$

is standardized apart by rewriting the second occurrence from left of X, of Y, and the occurrence of 3 in this order:

$$q(X) \leftarrow p(X, X_1, Z), q(Y, W), not\, r(Y_2, Y_1, V), dom(V), Z \neq 1, Y \neq W, X = X_1, Y = Y_1, Y_2 = 3. \tag{11}$$

Clearly, any rule r can be transformed into standardized apart form, which is equivalent under answer set semantics, in linear time; the order in which the steps are applied and the choice of variables and their occurrence does not matter. Notably, as a result, all arguments of non-relational atoms different from *dom* atoms are distinct variables; we do not need to standardize the arguments of *dom* literals apart, as both in the original domain D and in the abstract domain D, every *dom* atom always evaluates to true.

3.1.2. Lifted equality relation

As shown above, the (in)equality relations in the rules may cause issues in the abstraction process. To deal with them, we focus on rules r of the form

$$r : l \leftarrow B^{std}(r), \Gamma_{rel}(r) \tag{12}$$

where $B^{std}(r)$ are the non-relation atoms standardized apart, and Γ_{rel} consists of relation atoms that constrain the variables in $B^{std}(r)$. If r contains no relation, then $\Gamma_{rel} = \top$; an arbitrary rule r is easily rewritten to this form. We use $B^{std,+}$ (resp. $B^{std,-}$) to refer to positive (resp. negative) non-relational literals standardized apart.

The uncertainty that arises during the abstraction is caused by relation restrictions over non-singleton clusters $\widehat{d}$ (i.e., $|\widehat{d}| > 1$) or by negative literals mapped to non-singleton abstract literals. In order to address the uncertainty due to relation restrictions in the rules, we consider a notion of *relation types* with respect to the abstraction. For simplicity, we focus on a binary relation $\circ \in \{=, \neq\}$ and a relation part $\Gamma_{rel}(r)$ with at most one relation atom. Later in Section 3.3, we show how other forms of relations can be addressed.

When the relation $\circ$ is lifted to the abstract domain $\widehat{D}$, applying the relation to some $\hat{d}_1, \hat{d}_2 \in \widehat{D}$ with $\hat{d}_1 \circ \hat{d}_2$ may result in outcomes different than in the original domain, depending on the mapping, as we have seen in Example 3.6 and Table 1.

Table 2 shows the computation of the facts representing these types for a relation $\circ$. Type I, $\tau_{\mathrm{I}}^{\circ}$, and type II, $\tau_{\mathrm{II}}^{\circ}$, are the cases of no uncertainty; they hold for abstract elements $\hat{d}_1, \hat{d}_2$ if the relation $\hat{d}_1 \circ \hat{d}_2$ is true resp. false and if for every original elements x_1, x_2 mapped to $\hat{d}_1, \hat{d}_2$, respectively, that $x_1 \circ x_2$ is true resp. false in the original domain as well.

Example 3.9. Consider the mapping $m_1 = \{\{1\} \mapsto k_1, \{2, 3\} \mapsto k_2, \{4, 5\} \mapsto k_3\}$ in Table 1. The relation $k_1 = k_1$ holds and for any $x_1, x_2 \in k_1 = \{1\}$, $x_1 = x_2$ holds and type I applies for having $k_1 = k_1$ true. Further, $k_1 = k_2$ does not hold, i.e., $k_1 \neq k_2$, and for any $x \in k_1 = \{1\}$ and $y \in k_3 = \{2, 3\}$, $x = y$ does not hold and so type II applies for $k_1 = k_2$ false.

As for the relation $\neq$, we have that $k_1 \neq k_2$ holds true, and for $x \in k_1 = \{1\}$ and $y \in k_3 = \{2, 3\}$, $x \neq y$ holds true, so type I applies for having $k_1 \neq k_2$ true. Further, the relation $k_1 \neq k_1$ does not hold, and for any $x_1, x_2 \in k_1 = \{1\}$, $x_1 \neq x_2$ does not hold and type II applies for having $k_1 \neq k_1$ false.

As for type III and type IV in Table 2, they are the cause for uncertainty. Type III, $\tau^{\circ}_{\text{III}}$, (resp. Type IV, τ°_{IV}) holds for the abstract elements $\hat{d}_1, \hat{d}_2$ when $\hat{d}_1 \circ \hat{d}_2$ is true (resp. is not true), but for some x_1, x_2 that can be mapped to $\hat{d}_1, \hat{d}_2$, $x_1 \circ x_2$ does not hold true (resp. holds true) in the original domain.

Example 3.10 *(Example 3.9 ctd).* Consider again the mapping m_1 in Table 1. The relation $k_2 = k_2$ holds but for some $x_1, x_2 \in k_1 = \{2, 3\}$, $x_1 = x_2$ does not hold, e.g., $2 \neq 3$, thus type III applies. Further, $k_2 \neq k_2$ does not hold, i.e., $k_2 = k_2$, but for some $x, y \in k_2 = \{2, 3\}$, $x \neq y$ does hold, e.g., $2 \neq 3$ and so type IV applies.

If $\hat{d}_1 \circ \hat{d}_2$ holds for some $\hat{d}_1, \hat{d}_2 \in \widehat{D}$, type III is more common in practice in domain abstractions with clusters due to the standardization and as the relation $=$ often occurs, while type I occurs for singleton mappings (i.e., $|\hat{d}_1| = |\hat{d}_2| = 1$) or for the relation $\neq$. If $\hat{d}_1 \circ \hat{d}_2$ does not hold for some $\hat{d}_1, \hat{d}_2 \in \widehat{D}$, type II is common (again as the relation $=$), whereas type IV may occur for the relation $\neq$. In order to refer to the abstract relation atoms, we introduce the following notation.

Definition 3.3 *($\mathcal{T}_m$).* For an abstraction m, we let $\mathcal{T}_m$ be the set of all atoms $\tau^{\circ}_{\iota}(\hat{d}_1, \hat{d}_2)$ where $\iota \in \{\text{I}, \ldots, \text{IV}\}$ is the type of the binary relation $\hat{d}_1 \circ \hat{d}_2$ for m.

We remark that $\mathcal{T}_m$ is easily computed. Armed with these techniques, we now proceed to construct an abstract program for a given program Π and an abstraction mapping m, where we use $\mathcal{T}_m$ as facts.

Example 3.11 *(Example 3.9 ctd).* For the mapping m_1 shown in Table 1, we have the facts

$$\mathcal{T}_{m_1} = \{\tau^{=}_{\text{I}}(k_1, k_1),\ \tau^{=}_{\text{III}}(k_2, k_2),\ \tau^{=}_{\text{III}}(k_3, k_3),\ \tau^{=}_{\text{II}}(k_1, k_2),\ \tau^{=}_{\text{II}}(k_1, k_3),\ \tau^{=}_{\text{II}}(k_2, k_3),$$
$$\ldots,\ \tau^{\neq}_{\text{I}}(k_1, k_2),\ \tau^{\neq}_{\text{I}}(k_1, k_3),\ \tau^{\neq}_{\text{I}}(k_2, k_3), \ldots,\ \tau^{\neq}_{\text{II}}(k_1, k_1),\ \tau^{\neq}_{\text{IV}}(k_2, k_2),\ \tau^{\neq}_{\text{IV}}(k_3, k_3)\}.$$

3.2. Abstract program construction

By our analysis in Section 3.1, the basic idea to construct an abstract program Π' for a program Π with a domain mapping m is as follows. We either just abstract each atom in a rule, or in case of uncertainty due to domain abstraction, we guess rule heads to catch possible cases, or we treat negated literals by shifting their polarity depending on the abstract domain clusters.

We use an auxiliary fact $isSingleton(\hat{d})$ for the abstract domain elements $\hat{d} \in \widehat{D}$ to denote that $\hat{d}$ is a singleton cluster, i.e., $|m^{-1}(\hat{d})| = 1$. These atoms can also be used to represent whether an abstract atom $m(\alpha)$ is a singleton cluster, i.e., no other atom α' exists such that $m(\alpha') = m(\alpha)$: if every term $t \in arg(m(\alpha))$ satisfies $isSingleton(t)$, then $m(\alpha)$ is a singleton cluster, otherwise not.

Example 3.12 *(Example 3.1 ctd).* Consider the domain mapping $m_1 = \{\{1\} \mapsto k_1, \{2, 3\} \mapsto k_2, \{4, 5\} \mapsto k_3\}$. For the abstract domain, we have $isSingleton(k_1)$. For the literals, the singleton clusters are $a(k_1), c(k_1), d(k_1), e(k_1)$ and $b(k_1, k_1)$, while the remaining literals are non-singletons.

We remark that due to their definition, if either $\tau^{\circ}_{\text{III}}(\hat{d}_1, \hat{d}_2)$ or $\tau^{\circ}_{\text{IV}}(\hat{d}_1, \hat{d}_2)$ holds true for some $\hat{d}_1, \hat{d}_2 \in \widehat{D}$, this means that either $\hat{d}_1$ or $\hat{d}_2$ is a non-singleton cluster.

For ease of presentation and a systematic treatment of the issues that we have identified in Section 3.1, we shall first deal in Section 3.2.1 with programs under restrictions, which are then gradually lifted in Section 3.2.2 to arrive at arbitrary programs.

3.2.1. Restricted case

We first consider programs Π with rules that fulfill the following conditions:

(R-1) Unique negative literal. Each rule has at most one negative body literal.

(R-2) Unique relation atom. Each rules has no or a single, binary relation atom.

(R-3) No negative cycles. There are no negative cyclic dependencies between non-ground literals occurring in Π.

For rules r with no relation, i.e., $\Gamma_{rel} = \top$, we introduce for convenience a dummy relation $rel_{\top}(X, Y)$ that holds for all pairs of elements of the domain D and add in the body of r an atom $rel_{\top}(X, Y)$; in every lifting of $rel_{\top}$ from D to $\widehat{D}$ by

some mapping m, the relation type $\tau_{\mathrm{T}}^{rel_\top}(\hat{d}_1, \hat{d}_2)$ similar as in Table 2 will for all $\hat{d}_1, \hat{d}_2 \in \widehat{D}$ be true for type T = I and false for all other types. We then can assume that rules have a single binary relation atom $t_1 \circ t_2$ in the body.

We next define how to construct from a given rule r an abstract (set of) rule(s) r^m.

Definition 3.4 *(rule abstraction r^m).* Given a rule $r: \alpha \leftarrow B^{std}(r), t_1 \circ t_2$ as above and a domain mapping m, the set r^m contains the following rules:

(a) $m(\alpha) \leftarrow m(B^{std}(r)), m(t_1) \circ m(t_2), \tau_{\mathrm{I}}^{\circ}(m(t_1), m(t_2))$.
(b) $\{m(\alpha)\} \leftarrow m(B^{std}(r)), m(t_1) \circ m(t_2), \tau_{\mathrm{III}}^{\circ}(m(t_1), m(t_2))$.
(c) $\{m(\alpha)\} \leftarrow m(B^{std}(r)), m(t_1)\, \overline{\circ}\, m(t_2), \tau_{\mathrm{IV}}^{\circ}(m(t_1), m(t_2))$.
(d) For every $t \in arg(\alpha_i)$ where $B^{std,-}(r) = \{\alpha_i\}$:
  (i) $\{m(\alpha)\} \leftarrow m(B^{sh}_{\alpha_i}(r)), m(t_1) \circ m(t_2), not\ isSingleton(m(t))$.
  (ii) $\{m(\alpha)\} \leftarrow m(B^{sh}_{\alpha_i}(r)), m(t_1)\, \overline{\circ}\, m(t_2), \tau_{\mathrm{IV}}^{\circ}(m(t_1), m(t_2)), not\ isSingleton(m(t))$.

where $B^{sh}_{\alpha_i}(r) = B^{std,+}(r) \cup \{\alpha_i\}$.

In step (a), we have the case of no uncertainty due to abstraction. Steps (b) and (c) are for the cases of uncertainty due to the behavior of the relations. The head becomes a choice, and for case IV, we flip the relation, $\overline{\circ}$, in order to catch the case where the relation holds true (which actually is the reason for the uncertainty). No rules are added for case II, since the body of the rule will never be satisfied as the relation does not hold true in the abstract domain (similar as in the original domain). As for constraints (e.g., (6)), we note that $m(\bot) = \bot$. Consequently, in (a) the head is unchanged; as an optimization, all other steps (b)-(d) can be omitted, since the choice $\{\bot\}$ is ineffective (we always can choose that $\bot$ is false). Recall that the lifted relation $rel_\top$ will only have type I, thus for $rel_\top(t_1, t_2)$ only the rules in steps (a) and (d.i) in case must be added, where $rel_\top(m(t_1), m(t_2))$ and $\tau_{\mathrm{I}}^{rel_\top}(m(t_1), m(t_2))$ can be omitted as they are always true.

Example 3.13 *(ctd).* The abstract rules for (5) and (7) (the standardized apart version of (6)) become

$$e(X) \leftarrow c(X), a(Y), X \neq Y, \tau_{\mathrm{I}}^{\neq}(X, Y). \tag{13}$$
$$\{e(X)\} \leftarrow c(X), a(Y), X \neq Y, \tau_{\mathrm{III}}^{\neq}(X, Y). \tag{14}$$
$$\{e(X)\} \leftarrow c(X), a(Y), X = Y, \tau_{\mathrm{IV}}^{\neq}(X, Y). \tag{15}$$
$$\bot \leftarrow b(X, Y), e(X_1), X = X_1, \tau_{\mathrm{I}}^{=}(X, X_1). \tag{16}$$

Here the rule (14) can be omitted as $X \neq Y$, $\tau_{\mathrm{III}}^{\neq}(X, Y)$ is unsatisfiable (since $X \neq Y$ cannot have type III) and thus the rule body is unsatisfiable.

In step (d) of Definition 3.4, we grasp the uncertainty arising from negation by adding rules that shift the negative literal only if it shares arguments that are mapped to a non-singleton cluster.

Example 3.14 *(ctd).* The abstract rules for the rule (8) in Example 3.7 with a default negated literal will then be[3]

$$b(X, Y) \leftarrow a(X), not\ d(Y), dom(Y). \tag{17}$$
$$\{b(X, Y)\} \leftarrow a(X), d(Y), not\ isSingleton(Y). \tag{18}$$

We remark that simply omitting all default negated literals in the abstract rule is also possible. However, this would then cause to trigger choice rules in every case, thus resulting in more abstract answer sets than necessary, as shown below.

Example 3.15. Let the rule (8) in Example 3.7 contain in addition the relation $X \neq Y$:

$$b(X, Y) \leftarrow a(X), not\ d(Y), X \neq Y, dom(Y). \tag{19}$$

If instead of adding the rule

$$b(X, Y) \leftarrow a(X), not\ d(Y), X \neq Y, \tau_{\mathrm{I}}^{\neq}(X, Y). \tag{20}$$

we would omit $d(Y)$ and turn the head into a choice, thus have

[3] When the negated literal contains more than one argument, rules of form (18) will be added for each argument.

$$\{b(X,Y)\} \leftarrow a(X), X \neq Y, \mathit{not}\ \mathit{isSingleton}(Y), \mathit{dom}(Y). \tag{21}$$

then for the case where Y is a cluster while $X \neq Y$ holds true, the choice would unnecessarily trigger even if actually *not* $d(Y)$ was true. To see this, consider the mapping $m = \{\{1\} \to k_1, \{2,\ldots,5\} \to k_2\}$. The original program with the rule (19) has the answer set $I = \{a(1), a(3), d(5), c(2), b(1,2), b(1,3), b(1,4), b(3,1), b(3,2), b(3,4), e(2)\}$. Even though in the abstract program the fact $d(k_1)$ does not exist, the added choice rule (21) causes a guess on $b(k_2,k_1)$ which results in four spurious answer sets not containing $b(k_2,k_1)$, one of them being $I' = \{a(k_1), a(k_2), c(k_2), d(k_2), e(k_2),\ b(k_1,k_2), b(k_2,k_2)\}$. Adding instead the rule (20) in the abstract program avoids these spurious answer sets.

Distinguishing the different cases helps to avoid unnecessary answer sets when actually no uncertainty is caused. Thus, the abstract rules for (19) would be

$$b(X,Y) \leftarrow a(X), \mathit{not}\ d(Y), X \neq Y, \tau_{\mathrm{I}}^{\neq}(X,Y). \tag{22}$$

$$\{b(X,Y)\} \leftarrow a(X), \mathit{not}\ d(Y), X = Y, \tau_{\mathrm{IV}}^{\neq}(X,Y). \tag{23}$$

$$\{b(X,Y)\} \leftarrow a(X), d(Y), X \neq Y, \mathit{not}\ \mathit{isSingleton}(Y). \tag{24}$$

$$\{b(X,Y)\} \leftarrow a(X), d(Y), X = Y, \tau_{\mathrm{IV}}^{\neq}(X,Y), \mathit{not}\ \mathit{isSingleton}(Y). \tag{25}$$

where again the rule for case (b) is omitted ($X \neq Y, \tau_{\mathrm{III}}^{\neq}(X,Y)$ is unsatisfiable).

Semantically, the rules added in steps (a)-(b) of Definition 3.4 are to ensure that each model I of Π carries over to a model $m(I)$ of the rule abstraction, as either the original rule is kept or changed to a choice rule. As regards an answer set I, steps (c)-(d) serve to catch in particular the cases that may violate the minimality in the abstraction due to a negative literal or a relation over non-singleton clusters. The abstract program is now as follows.

Definition 3.5 *(abstract program Π^m, restricted case).* Given a (standardized apart) program Π as above and a domain abstraction m, the abstract program for m, denoted Π^m, consists of the rules

$$\Pi^m = \bigcup_{r:\ \alpha \leftarrow B^{std}(r), t_1 \circ t_2 \in \Pi} r^m \ \cup\ \{x. \mid x \in \mathcal{T}_m\} \ \cup\ \{m(p(\vec{c})). \mid p(\vec{c}). \in \Pi\} \cup \{\mathit{isSingleton}(\hat{d}) \mid |m^{-1}(\hat{d})| = 1\}. \tag{26}$$

Notably, the construction of Π^m is modular, rule by rule. The following result states that this abstraction works.

Theorem 3.1 *(restricted program abstraction). Let m be a domain mapping of a (standardized apart) program Π under the above assumptions (R-1)–(R-3). Then for every $I \in AS(\Pi)$, it holds that $m(I) \cup \mathcal{T}_m \in AS(\Pi^m)$.*

The proof of this result proceeds along the intuition above more formally, where for showing that the abstraction $m(I)$ of an answer set I is a minimal model also recursive dependencies through negative literals ought to be considered. Since no negative cyclic dependencies between ground literals exist, counterexamples to minimality along such dependencies can be excluded.

3.2.2. General case

We now describe how to remove the restrictions (R-1)–(R-3) on programs from above.

(G-1) Multiple negative literals. If rule r has multiple negative literals, i.e., $|B^-(r)|>1$, we shift each negative literal that shares arguments that are mapped to a non-singleton cluster. Thus, instead of shifting one literal $\alpha_i \in B^-(r)$, we consider shifting multiple literals $L \subseteq B^-(r)$ at a time and all combinations of (non-)shifting the literals in $B^-(r)$.

Definition 3.6 *(treating multiple negative literals).* Step (d) of Definition 3.4 is modified as

(d) For each $L = \{\alpha_1, \ldots, \alpha_n\} \subseteq B^{std,-}(r)$, $n \geq 1$ and each $t^1, \ldots, t^n$ where $t^i \in arg(\alpha_i)$, $i = 1, \ldots, n$:
  (i) $\{m(\alpha)\} \leftarrow m(B_L^{sh}(r)), m(t_1) \circ m(t_2), \mathit{not}\ \mathit{isSingleton}(m(t^1)), \ldots, \mathit{not}\ \mathit{isSingleton}(m(t^n))$.
  (ii) $\{m(\alpha)\} \leftarrow m(B_L^{sh}(r)), m(t_1)\ \overline{\circ}\ m(t_2), \tau_{\mathrm{IV}}^{\circ}(m(t_1), m(t_2)), \mathit{not}\ \mathit{isSingleton}(m(t^1)), \ldots, \mathit{not}\ \mathit{isSingleton}(m(t^n))$.

where $B_L^{sh}(r) = B^{std,+}(r) \cup L, \mathit{not}\ (B^{std,-}(r) \setminus L)$.

Shifting all the negated atoms L is for the case where all these atoms give rise to non-singleton clusters, which occurs when for each atom α_i some argument t^i is mapped to a non-singleton cluster; in the original domain, the negative rule body on L could thus evaluate to true and the rule could possible fire. The rules added for all arguments $t^1, \ldots, t^n$ ensure that all cases are covered. Steps (d.i) and (d.ii) coincide with steps (d.i) and (d.ii) of Definition 3.4, respectively.

Table 3
Cases for lifting an n-ary relation rel'.

$\tau_{\mathrm{I}}^{rel'}(\hat{d}_1,\ldots,\hat{d}_n)$:	$rel'(\hat{d}_1,\ldots,\hat{d}_n) \wedge \forall x_1 \in m^{-1}(\hat{d}_1),\ldots,\forall x_n \in m^{-1}(\hat{d}_n).\, rel'(x_1,\ldots,x_n)$
$\tau_{\mathrm{II}}^{rel'}(\hat{d}_1,\ldots,\hat{d}_n)$:	$\neg rel'(\hat{d}_1,\ldots,\hat{d}_n) \wedge \forall x_1 \in m^{-1}(\hat{d}_1),\ldots,\forall x_n \in m^{-1}(\hat{d}_n).\, \neg rel'(x_1,\ldots,x_n)$
$\tau_{\mathrm{III}}^{rel'}(\hat{d}_1,\ldots,\hat{d}_n)$:	$rel'(\hat{d}_1,\ldots,\hat{d}_n) \wedge \exists x_1 \in m^{-1}(\hat{d}_1),\ldots,\exists x_n \in m^{-1}(\hat{d}_n).\, \neg rel'(x_1,\ldots,x_n)$
$\tau_{\mathrm{IV}}^{rel'}(\hat{d}_1,\ldots,\hat{d}_n)$:	$\neg rel'(\hat{d}_1,\ldots,\hat{d}_n) \wedge \exists x_1 \in m^{-1}(\hat{d}_1),\ldots,\exists x_n \in m^{-1}(\hat{d}_n).\, rel'(x_1,\ldots,x_n)$

Example 3.16. Consider the rule

$$d(X) \leftarrow not\ c(X), not\ a(X_1), X = X_1, dom(X), dom(X_1).$$

The constructed non-ground abstract rules following step (d.i) of Definition 3.6 will be

$$\{d(X)\} \leftarrow c(X), not\ a(X_1), X = X_1, not\ isSingleton(X), dom(X_1).$$
$$\{d(X)\} \leftarrow not\ c(X), a(X_1), X = X_1, not\ isSingleton(X_1), dom(X).$$
$$\{d(X)\} \leftarrow c(X), a(X_1), X = X_1, not\ isSingleton(X), not\ isSingleton(X_1).$$

Step (d.ii) is similarly applied.

(G-2) Multiple relation atoms. A simple approach to handle multiple relations, i.e.,

$$\Gamma_{rel} = t_{1,1} \circ_1 t_{2,1}, .., t_{1,k} \circ_k t_{2,k}, k > 1, \tag{27}$$

is to view it as an atom of an $2k$-ary relation $rel'(t_{1,1}, t_{2,1}, .., t_{1,k}, t_{2,k})$. The atom $rel'(t_{1,1}, t_{2,1}, .., t_{1,k}, t_{2,k})$ is true if all relations $t_{1,1} \circ_1 t_{2,1}, .., t_{1,k} \circ_k t_{2,k}$ are true, and it is false, i.e., $\neg rel'(t_{1,1}, t_{2,1}, .., t_{1,k}, t_{2,k})$ is true, if some relation $t_{1,i} \circ_i t_{2,i}$ is false. The abstract version of such rel' and the cases I-IV are then lifted from x_1, x_2 to $x_1, .., x_n$ as in Table 3.

Example 3.17. For $\Gamma_{rel} = (X_1{=}X_2,\ X_3{=}X_4)$, we use a new relation $rel'(X_1, X_2, X_3, X_4)$. If for abstract values $\hat{d}_1, .., \hat{d}_4$, $\hat{d}_1 = \hat{d}_2 \wedge \hat{d}_3 = \hat{d}_4$ holds, then we have type τ_{I} when all $\hat{d}_i$ are singleton clusters and type τ_{III} when some $\hat{d}_i$ is non-singleton; otherwise (i.e., $\neg rel'(\hat{d}_1, \hat{d}_2, \hat{d}_3, \hat{d}_4)$ holds) type τ_{II} applies.

(G-3) Cyclic negative dependencies. Rules which are involved in a negative cyclic dependency need special consideration.

Example 3.18 *(Example 3.1 ctd).* Now consider adding the following rules to the program Π and removing the facts $d(5), c(2)$.

$$c(X) \leftarrow not\ d(X), dom(X). \tag{28}$$
$$d(X) \leftarrow not\ c(X), dom(X). \tag{29}$$

These rules create further answer sets for Π containing different appearances of c and d. The abstract rules will be as follows.

$$c(X) \leftarrow not\ d(X), dom(X). \tag{30}$$
$$\{c(X)\} \leftarrow d(X), not\ isSingleton(X). \tag{31}$$
$$d(X) \leftarrow not\ c(X), dom(X). \tag{32}$$
$$\{d(X)\} \leftarrow c(X), not\ isSingleton(X). \tag{33}$$

Now consider an answer set I of Π which contains $d(1), c(2), d(3), c(4)$, and $d(5)$, and the naive mapping $m_2 = \{\{1,\ldots,5\} \mapsto k\}$. The mapping $\hat{I} = m(I)$ of the answer set I contains $c(k)$ and $d(k)$. Although $\hat{I}$ is a model of $(\Pi^m)^I$, either $c(k)$ or $d(k)$ is unfounded; hence $\hat{I}$ is not minimal, i.e., not an answer set of Π^m. The reason is that the negative cyclic dependency (i.e., "choice") of c- and d-atoms does not occur for $c(k)$ and $d(k)$ in the constructed Π^m.

To resolve this, the literals of Π that are involved in a negative cycle are treated specially. We can lift the restriction (R-3) in the restricted case as follows.

Definition 3.7 *(treating negative cyclic dependency).* Suppose that $L_{c_1}, \ldots, L_{c_l}$ are all independent sets of literals involved in a negative cyclic dependency. If $l > 0$, item (d) of Definition 3.4 is changed as follows:

(d) For every $t \in arg(\alpha_i)$ where $B^{std,-}(r) = \{\alpha_i\}$, for every $j = 1, \ldots, l$:

(i) $\{m(\alpha)\} \leftarrow m(B^{sh}_{\alpha_i,L_{c_j}}(r)), m(t_1) \circ m(t_2), \mathit{not}\ \mathit{isSingleton}(m(t))$.

(ii) $\{m(\alpha)\} \leftarrow m(B^{sh}_{\alpha_i,L_{c_j}}(r)), m(t_1) \mathbin{\overline{\circ}} m(t_2), \tau^{\circ}_{\mathrm{IV}}(m(t_1), m(t_2)), \mathit{not}\ \mathit{isSingleton}(m(t))$.

where

$$B^{sh}_{\alpha_i,L_{c_j}}(r) = \begin{cases} B^{std,+}(r) \cup \{\alpha_i\} & \text{if } \{\alpha_i, \alpha\} \not\subseteq L_{c_j}, \\ B^{std,+}(r) & \text{if } \{\alpha_i, \alpha\} \subseteq L_{c_j}. \end{cases}$$

In the generalization of step (d) of Definition 3.4, the newly defined $B^{sh}_{\alpha_i,L_{c_j}}(r)$ eliminates the atoms α_i that are involved in a negative cycle with the head α of the rule r, i.e., $\{\alpha_i, \alpha\} \subseteq L_{c_j}$ for some j, from the body instead of shifting their polarity. Since the sets $L_{c_1}, \ldots, L_{c_l}$ are independent, a rule can be involved with at most one cyclic set L_{c_j} of literals, and thus the cycles can be treated one at a time.

Example 3.19 *(ctd).* For the program Π consisting of the rules (4)-(6) and (28)-(29), the abstract non-ground rules are

$$c(X) \leftarrow \mathit{not}\ d(X), \mathit{dom}(X). \tag{34}$$

$$\{c(X)\} \leftarrow \mathit{not}\ \mathit{isSingleton}(X), \mathit{dom}(X). \tag{35}$$

$$d(X) \leftarrow \mathit{not}\ c(X), \mathit{dom}(X). \tag{36}$$

$$\{d(X)\} \leftarrow \mathit{not}\ \mathit{isSingleton}(X), \mathit{dom}(X). \tag{37}$$

$$b(X,Y) \leftarrow a(X), d(Y). \tag{38}$$

$$e(X) \leftarrow c(X), a(Y), X \neq Y, \tau^{\neq}_{\mathrm{I}}(X,Y). \tag{39}$$

$$\{e(X)\} \leftarrow c(X), a(Y), X = Y, \tau^{\neq}_{\mathrm{IV}}(X,Y). \tag{40}$$

$$\bot \leftarrow b(X,Y), e(X_1), X = X_1, \tau^{=}_{\mathrm{I}}(X,Y). \tag{41}$$

For the mapping $m = \{\{1,\ldots,5\} \mapsto k\}$, the facts $\{a(1), a(3)\}$ in Π get lifted to $\{a(k)\}$ and the type facts become $\mathcal{T}_m = \{\tau^{=}_{\mathrm{III}}(k,k), \tau^{\neq}_{\mathrm{IV}}(k,k)\}$. Note that the fact $\{\mathit{isSingleton}(k)\}$ is not true. The abstract program Π^m consists of all the abstract rules and the mentioned facts. Notice that when the rules are grounded to the relation type facts $\mathcal{T}_m$, only the rules (34)-(38) and (40) remain to be used for the answer set computation.

(G-1)–(G-3), i.e., arbitrary programs. Lifting (R-2) to (G-2) can be easily done jointly with lifting (R-1) to (G-1) and (R-3) to (G-3), respectively. The joint lifting of (R-1) and (R-3) is achieved by a generalization of Definition 3.4 that combines the ideas of Definitions 3.6 and 3.7.

Definition 3.8 *(treating cyclic dependency with multiple negative literals).* Suppose that $L_{c_1}, \ldots, L_{c_l}$ are all independent sets of literals involved in a negative cyclic dependency. Then item (d) of Definition 3.4 is changed as in Definition 3.6 if $l = 0$, and otherwise as follows:

(d) For each $L = \{\alpha_1, \ldots, \alpha_n\} \subseteq B^{std,-}(r)$, $n \geq 1$, and $t^1, \ldots, t^n$ where $t^i \in \mathit{arg}(\alpha_i)$, $i = 1, \ldots, n$, and for every $j = 1, \ldots, l$:

(i) $\{m(\alpha)\} \leftarrow m(B^{sh}_{L,L_{c_j}}(r)), m(t_1) \circ m(t_2), \mathit{not}\ \mathit{isSingleton}(m(t^1)), \ldots, \mathit{not}\ \mathit{isSingleton}(m(t^n))$.

(ii) $\{m(\alpha)\} \leftarrow m(B^{sh}_{L,L_{c_j}}(r)), m(t_1) \mathbin{\overline{\circ}} m(t_2), \tau^{\circ}_{\mathrm{IV}}(m(t_1), m(t_2)), \mathit{not}\ \mathit{isSingleton}(m(t^1)), \ldots, \mathit{not}\ \mathit{isSingleton}(m(t^n))$.

where

$$B^{sh}_{L,L_{c_j}}(r) = \begin{cases} B^{std,+}(r) \cup L, \mathit{not}\ B^{std,-}(r) \setminus L & \text{if } \alpha \notin L_{c_j} \text{ or } L_{c_j} \cap L = \emptyset, \\ B^{std,+}(r) \cup (L \setminus L_{c_j}), \mathit{not}\ B^{std,-}(r) \setminus L & \text{if } \alpha \in L_{c_j} \text{ and } L_{c_j} \cap L \neq \emptyset; \end{cases} \tag{42}$$

that is, the negative literals in L get their polarity shifted if they do not occur in some L_{c_j} with the head α of rule r, otherwise they are omitted.

As mentioned in Example 3.15, simply omitting all negated literals is also possible, though can cause more spurious abstract answer sets. Thus in Definition 3.6 we considered the positive shift of the negated atoms by considering all combinations. As for negated atoms that are involved in a negative cycle with the rule head, Example 3.18 showed that a positive shift on them can prevent over-approximation. Definition 3.8 combines these two insights, and ensures that the atoms in the subset L of $B^{std,-}(r)$ are omitted from shifting if they are involved in a cycle with the head, while the remaining atoms in L are shifted. As we consider all subsets L of $B^{std,-}(r)$, we also construct abstract rules where these literals remain untouched.

Example 3.20. Consider the rules

$$\begin{aligned} &c(X) \leftarrow not\ d(X), dom(X).\\ &d(X) \leftarrow not\ c(X), not\ a(X_1), X = X_1, dom(X), dom(X_1).\\ &a(X) \leftarrow not\ b(X), dom(X).\\ &b(X) \leftarrow not\ a(X), dom(X). \end{aligned} \tag{43}$$

with independent cycles $L_{c_1} = \{d(X), c(X)\}$ and $L_{c_2} = \{a(X), b(X)\}$ The constructed non-ground abstract rules of (43) following step (d.i) of Definition 3.8 will be

$$\{d(X)\} \leftarrow not\ a(X_1), X = X_1, not\ isSingleton(X), dom(X), dom(X_1). \tag{44}$$

$$\{d(X)\} \leftarrow c(X), not\ a(X_1), X = X_1, not\ isSingleton(X), dom(X_1). \tag{45}$$

$$\{d(X)\} \leftarrow not\ c(X), a(X_1), X = X_1, not\ isSingleton(X_1), dom(X). \tag{46}$$

$$\{d(X)\} \leftarrow a(X_1), X = X_1, not\ isSingleton(X), not\ isSingleton(X_1), dom(X). \tag{47}$$

$$\{d(X)\} \leftarrow c(X), a(X_1), X = X_1, not\ isSingleton(X), not\ isSingleton(X_1). \tag{48}$$

Observe that (44), constructed with $B^{sh}_{\{c(X)\},L_{c_1}}$, is a stronger rule than (45), constructed with $B^{sh}_{\{c(X)\},L_{c_2}}$, due to the omission of $c(X)$. The rule (46) gets constructed with $B^{sh}_{\{a(X_1)\},L_{c_1}}$ and $B^{sh}_{\{a(X_1)\},L_{c_2}}$, where $a(X_1)$ does not get omitted, since it is not involved in a negative cycle with $d(X)$. The rule (47) is an outcome of $B^{sh}_{\{a(X_1),c(X)\},L_{c_1}}$ where $c(X)$ gets omitted, while $a(X_1)$ is shifted, which is a stronger rule than (48) constructed with $B^{sh}_{\{a(X_1),c(X)\},L_{c_2}}$.

The non-subsumed rules constructed following step (d.i) of Definition 3.8 then become the rules (44), (46), (47) together with

$$\begin{aligned} &\{c(X)\} \leftarrow not\ isSingleton(X), dom(X).\\ &\{a(X)\} \leftarrow not\ isSingleton(X), dom(X).\\ &\{b(X)\} \leftarrow not\ isSingleton(X), dom(X). \end{aligned}$$

Step (d.ii) is similarly applied.

We now have all bits in order to define the abstract version of an arbitrary program Π.

Definition 3.9 *(abstract program Π^m, general case).* Given a (standardized apart) program Π and a domain abstraction m, the abstract program for m, denoted Π^m, consists of the rules as in (26), where r^m for each $r \in \Pi$ is as in Definition 3.8 (the modified version of Definition 3.4), and multiple relations as in (27) are replaced by a relation rel' as described in the respective case **(G-2)**.

Note that for programs that fulfill the restrictions (R-1)–(R-3), the Definitions 3.9 and 3.5 coincide, and thus Π^m is well-defined. The main result of this section is then as follows.

Theorem 3.2 *(general program abstraction). Let m be a domain mapping of a (standardized apart) program Π. Then for every $I \in AS(\Pi)$, the abstract interpretation $\hat{I} = m(I) \cup \mathcal{T}_m$ is an answer set of Π^m.*

This result is shown by an extension of the proof of Theorem 3.1, in which the more general conditions are taken into account.

Example 3.21 *(ctd).* The constructed abstract program Π^m has the five answer sets $\hat{I}_1 = \{a(k), d(k), b(k,k)\}$, $\hat{I}_2 = \{a(k), c(k)\}$, $\hat{I}_3 = \{a(k), c(k), d(k), b(k,k)\}$, $\hat{I}_4 = \{a(k), c(k), e(k)\}$, $\hat{I}_5 = \{a(k), c(k), d(k), e(k), b(k,k)\}$. Furthermore, for every answer set I of Π, $m(I)$ is an answer set of Π^m.

The abstraction yields in general an over-approximation of the answer sets of a program. The notion of spurious and concrete answer sets amounts to the following.

Definition 3.10 *(cf. Definition 2.5).* An abstract answer set $\hat{I} \in AS(\Pi^m)$ is *concrete* if there exists an answer set $I \in AS(\Pi)$ such that $\hat{I} = m(I) \cup \mathcal{T}_m$, else it is *spurious*.

A spurious abstract answer set has no corresponding concrete answer set.

Example 3.22 *(ctd).* The abstract answer sets $\hat{I}_2 = \{a(k), c(k)\}$ and $\hat{I}_3 = \{a(k), c(k), d(k), b(k,k)\}$ are spurious.

3.2.3. Abstract program size vs. over-approximation quality

The abstract program Π^m in Definition 3.9 can in general be exponentially larger than the original program Π, owing to the fact that dealing with multiple negative literals may introduce for every subset $L \subseteq B^{std,-}(r)$ of the negative literals in a rule r some rules in item (d) of Definition 3.8 respectively 3.6. On the other hand, multiple negative cycles do not cause an exponential blowup, since the number of independent negative cycles is bounded by the number of atoms that occur in the program Π. The items (a)–(c) of Definition 3.8 introduce few rules, which are akin to the original rules and preserve in a way their structure.

This blowup seems to be unpleasant, both from a cognitive and a computational perspective. However, it is mitigated by the following observations. First, the rules in item (d) of Definition 3.8 are generated systematically, and one may view them as a subprogram that is expanded on demand, if the user wants to inspect it; this can be similarly exploited for evaluation algorithms that avoid exponential space consumption (see Section 3.5). Second, in many cases the number of negative literals in a rule will be bounded by a constant, which means that no exponential blowup happens. Furthermore, grounding the rules to the relation type facts $\mathcal{T}_m$ may remove many of them. The size of the abstract program may also be kept smaller at the price of a weaker over-approximation. Specifically, we may in Definition 3.8 replace each negative cycle L_{c_j} by an arbitrary superset $S \supseteq L_{c_j}$ of literals in Π, while the resulting abstract program Π^m is still an over-approximation of Π. For example, we may choose $S = L_{c_1} \cup \cdots \cup L_{c_l}$, i.e., merge all negative cycles into one set, which may save half of the rules in item (d) of Definition 3.8. As many programs in practice, including those we considered here, do not have multiple negative cycles, we obtain the same abstraction. In fact, it can be seen that the latter also holds for multiple negative cycles (see Appendix A.1). If we let S be the set of all literals in the program Π, then (42) amounts to replacing systematically negated atoms $not\,\alpha_i$ by cluster information $not\,isSingleton(t^i)$. However, in the worst case, for an unbounded number of negative literals this still incurs an exponential blowup of Π^m.

We may avoid this by simply dropping in item (d) from rule r all negative literals, without adding further auxiliary literals; this still results in an over-approximation, at the price of further spurious answer sets. For example, consider the simple program $a(X) \leftarrow not\ d(X), dom(X).$ for domain $\{1,2,3\}$ with the fact $d(3)$. If in the abstract program we omit $d(X)$, then a choice on $a(X)$ will always occur, no matter if $d(3)$ is mapped to a singleton or not. Having further auxiliary atoms to distinguish this case then becomes useful.

Furthermore, we can reduce the number of rules in item (d) by using auxiliary predicates $hasCluster_k(X_1, \ldots, X_k)$ of arity $i \geq 1$, which express that some argument X_i, $1 \leq i \leq k$, is a non-singleton cluster; instead of using $not\ isSingleton(m(t^i))$, we then simple add for $\alpha_i = p(s^1, .., s^k)$ the atom $hasCluster_k(s^1, \ldots, s^k)$ to the rule bodies.

Another possibility is to simplify (for a chosen set S of literals as above) the program Π^m by eliminating subsumed rules as in Example 3.20, or by replacing multiple rules with other rules such that the answer sets of the program are not affected; e.g., one may think of replacing the rules in (b) and (d.i) (resp. (c) and (d.ii)) of Definition 3.4, when some atom $\alpha_i \in B^{std,-}(r)$ exists, by a merged rule from which the atom α_i is removed, for suitable terms t (when t is among t_1 and t_2 of $t_1 \circ t_2$, for instance). To this end, program rewriting and optimization techniques for ASP could be exploited (see Section 7.3.1 for more information). However this could change the structure of the resulting program significantly, such that it may be more difficult for the user to understand the working of the abstraction program, and obtaining intuitive explanations for spuriousness can be more difficult than when using the systematic approach. Furthermore, extending the approach to more language constructs might be more difficult to accomplish. Exploring the tradeoff between semantic accuracy of over-approximation, the size of the abstract program in the space of possible choices for supersets S, and possible structure-preserving optimizations is an interesting issue but beyond this article.

3.3. Syntactic extensions and further considerations

3.3.1. Other forms of relations

It is customary to use in ASP programs other relations apart from $=, \neq$ such as comparison $<, \leq$ or non-binary relations such as addition $X + Y = Z$ or multiplication $X * Y = Z$. ASP solvers support them in the input syntax as *built-in relations*, which are typically pre-evaluated during program grounding.

A simple way to treat such relations is as follows: (1) rewrite the relations by adding instead auxiliary atoms to represent them, (2) standardize apart the auxiliary atom arguments similarly as the remaining atoms, and (3) add to the original program facts of the auxiliary atom to show for which domain elements the relation holds true. In the abstraction procedure, the facts added will be lifted to the abstract domain, and the abstraction is handled over relations for the arguments which were standardized apart. We illustrate this on a small example.

Example 3.23. Consider the rule

$$b(X, Y) \leftarrow a(X), d(Y), X + 1 = Y.$$

For the addition relation, an auxiliary atom $plusOne(X, Y)$ is introduced by adding the facts $plusOne(1, 2)$, $plusOne(2, 3)$, $plusOne(3, 4)$, $plusOne(4, 5)\}$ to Π stating on which domain elements this relation holds.

The respective rule gets standardized apart into the following form.

$$b(X, Y) \leftarrow a(X), d(Y), plusOne(X_1, Y_1), X{=}X_1, Y{=}Y_1.$$

Fortunately, standardizing shared arguments in frequently used built-in atoms $X \leq Y$, $X < Y$ etc. apart, which introduces new variables and auxiliary atoms, can be avoided; to this end, the relation types in Table 2 may be extended to the relations $\circ \in \{=, \neq, <, \leq, >, \geq\}$ directly.

Respecting the order relation. If the original domain D has an order relation among its elements, i.e., $x_1 \circ x_2$ where $\circ \in \{<, \leq\}$, then $\circ$ should be defined in $\widehat{D}$ such that $\hat{d}_1 \circ \hat{d}_2$ can be evaluated for abstract values $\hat{d}_1$ and $\hat{d}_2$. Furthermore, the abstraction mapping should respect the ordering of elements to avoid unnecessary uncertainty.

Example 3.24 *(ctd).* Consider the mapping $m' = \{\{4\}\mapsto k_1, \{1, 5\}\mapsto k_2, \{2, 3\}\mapsto k_3\}$, which does not allow to respect the usual ordering $<$ in the abstraction: as $1 < 4 < 5$ we would need an ordering $k_2 < k_1 < k_2$ which is not possible (even resorting to a non-strict ordering $\leq$ would fail). While the relation types can still be defined for m', they will for $<$ be mostly of type III and IV, resulting in many uncertainties.

3.3.2. Strong negation and function symbols

Our abstraction method can also be applied to programs that contain strongly ("classically") negated atoms $\neg a$. The simple way to achieve this is to apply the traditional transformation where each strongly negated atom $\neg p(t_1, \ldots, t_n)$ is replaced by an atom $neg_p(t_1, \ldots, t_n)$ where neg_p is a fresh auxiliary predicate and a constraint "$\bot \leftarrow p(t_1, \ldots, t_n), neg_p(t_1, \ldots, t_n)$." is added to the program [57].

For programs with uninterpreted function symbols, auxiliary atoms that emulate terms containing function symbols with new constant symbols can be used, similarly as discussed in [41, Section 6]. For illustration, the rule $p(f(f(X))) \leftarrow q(X)$ can be rewritten as $p(Y) \leftarrow q(X), aux_f(X, U), aux_f(U, V)$ where informally the predicate $aux_f(c_1, c_2)$ links a constant symbol c_1 representing a term t_1 to a constant symbol c_2 that represents $f(t_1)$. Nested function terms can then be represented, as in the example rule, using multiple atoms. The predicate aux_f is precomputed and provided as facts. Notably, ASP programs are generally evaluated on a finite grounding of the input program [3]; hence the potentially infinite Herbrand universe does not prevent one to apply this method to such programs.

3.3.3. Treating choice rules and cardinality constraints

So far, we have considered choice rules as a shorthand for two ordinary rules (which is the standard definition of the semantics). It make sense, however, to consider them as primitive constructs with dedicated treatment to achieve more structure preservation. To this end, choice rules are treated by ensuring that the body is abstracted and the choice over the abstracted head is kept.

Definition 3.11. Given a choice rule $r: \{\alpha\} \leftarrow B^{std}(r), t_1 \circ t_2$ and a domain mapping m, the set r^m contains the rules of Definition 3.4 for steps (b)-(d), and for step (a), it contains

$$\{m(\alpha)\} \leftarrow m(B^{std}(r)), m(t_1) \circ m(t_2), \tau_{\mathrm{I}}^{\circ}(m(t_1), m(t_2)).$$

Cardinality constraints and conditional literals are further common syntactic extensions [117]; in particular, $i_\ell\{a(X) : b(X)\}i_u$ is true whenever at least i_ℓ and at most i_u instances of $a(X)$ subject to $b(X)$ are true. Choice rules that involve cardinality constraints, e.g., $n_1 \leq \{\alpha\} \leq n_2 \leftarrow B(r)$, cannot immediately be treated similarly. Lifting the cardinality constraints analogously to the abstract rule causes to force the occurrence of abstract atoms for ensuring the lower bound.

Example 3.25 *(ctd).* Consider instead of (4) the rule

$$2 \leq \{b(X, Y) : d(Y)\} \leq 4 \leftarrow a(X). \tag{49}$$

which gets lifted to the same abstract rule. However, for the mapping $m = \{\{1, 2, 3, 4, 5\}\mapsto k\}$, if $a(k)$ and $d(k)$ holds true, this would cause to have $b(k, k)$ hold true and no other atoms with the same predicate. Thus, the lower bound cannot be satisfied, causing the abstract program to become unsatisfiable.

The issue arises from the fact that if the atom in the choice head is involved in some non-singleton cluster, then multiple original atoms may be mapped to it, thus still satisfying the lower bound constraint in the original program. Such choice rules can be treated by modifying the lower bounds in the abstract program and adding a constraint to ensure that the original lower bound is met if the atom is only involved with singleton clusters.

Definition 3.12. Given a rule $r:\ n_1 \leq \{\alpha\} \leq n_2 \leftarrow B^{std}(r), t_1 \circ t_2$, in the abstraction procedure the choice head is changed to $\{m(\alpha)\} \leq n_2$, and an additional constraint of the following form is added.

$$\bot \leftarrow \{m(\alpha) : isSingleton(m(s^1)), \ldots, isSingleton(m(s^k))\}{<}n_1, B^{std}(r), m(t_1) \circ m(t_2), \tau_1^\circ(m(t_1), m(t_2)), \tag{50a}$$

$$\{m(\alpha) : not\ isSingleton(m(s^1)); \ldots; m(\alpha) : not\ isSingleton(m(s^k))\} < 1. \tag{50b}$$

where $\alpha = p(t^1, \ldots, t^n)$ and $s^1, \ldots, s^k$ are the terms among $t^1, \ldots, t^n$ that are local variables in r, i.e., variables not occurring in $B^{std}(r), t_1 \circ t_2$.

The idea with the additional constraint is to ensure that if the lower bound n_1 is not satisfied through literals mapped to singleton clusters (50a), then some literal with a non-singleton cluster (50b) should also occur.

Example 3.26 *(ctd).* Instead of lifting the choice rule as in Example 3.25, we add the following abstract rules:

$$\{b(X,Y) : d(Y)\} \leq 4 \leftarrow a(X).$$

$$\bot \leftarrow \{b(X,Y) : isSingleton(Y)\} < 2, a(X),$$

$$\{b(X,Y) : not\ isSingleton(Y), dom(Y)\} < 1.$$

This way there is no lower bound on the number of occurrences of $b(X,Y)$ that causes unsatisfiability at the abstract program. Furthermore, for the mapping $m = \{\{1\}\mapsto k_1, \{2,3,4,5\}\mapsto k_2\}$, for an answer set containing $b(k_1,k_1)$ the constraint ensures it also contains $b(k_1,k_2)$ so that the original lower bound is met.

By taking numeric information about cluster sizes into account, a more fine-grained treatment of the lower bound n_1 is possible. On the one hand, one can eliminate in more cases than those captured by the constraint (50a)-(50b) abstract models for which an overestimate of the number of atoms in the original answer set does not exceed n_1, where the overestimate is computed using aggregates and size of all clusters. In a dual approach, we can adjust n_1 to an underestimate $n_1' \leq n_1$ of the number of atoms that must be present in the abstract answer set, depending on the size of the largest cluster of an abstract constant used to instantiate a local variable (in the example, Y in $d(Y)$). To illustrate, if in (49) the lower bound would be 5 and variable Y would be instantiated with abstract constants of maximum cluster size 4, then we would know that we need at least two of the abstractly instantiated atoms $b(X,Y)$ to satisfy the concrete lower bound of 5. In general, we can adapt the lower bound n_1 in the abstract version of the cardinality constraint to $n_1' = \lceil n_1/mcs^{nlv} \rceil$ where mcs $(= mcs(m,r))$ is the maximum cluster size of an abstract constant in the mapping m that is used to instantiate a local variable in r, and nlv $(= nlv(r))$ is the number of all local variables. In the example, the computation adapts the lower bound $n_1 = 5$ to $n_1' = \lceil 5/(4^1) \rceil = 2$. However, these adaptations depend on a concrete mapping m, and the numbers have to be provided with (non-abstracted) auxiliary atoms or computed by a (non-abstracted) subprogram. For space reasons, we omit here working out concrete encoding techniques.

3.3.4. Concreteness with projection

Usually the problem encodings contain auxiliary atoms that are insignificant for solutions. When constructing the abstract program, such auxiliary atoms are treated the same, by introducing choices whenever there is an uncertainty. However, this causes many spurious guesses over the auxiliary atoms, and making sure that the abstract answer set is concrete w.r.t. all of these atoms becomes too ambitious, as encountering a concrete abstract answer set among many spurious ones is more difficult. For this reason, we consider a projected notion of determining concreteness of an abstract answer set by only focusing on a certain set of atoms.

Definition 3.13. For a set A of atoms, an abstract answer set $\hat{I} \in AS(\Pi^m)$ is *concrete w.r.t. A* if $\hat{I}|_{\hat{A}} = m(I|_A) \cup \mathcal{T}_m$ for an answer set $I \in AS(\Pi)$, where $\hat{A} = m(A)$.

Example 3.27. Consider a modified instance of graph coloring where the isolated nodes are connected as shown in Fig. 5. For the abstraction, the abstract coloring is spurious as the nodes in the cluster $\{4,5,6\}$ cannot all be colored to *red* in the original graph due to the edges. However, the abstract coloring is concrete w.r.t. the nodes $\{1,2,3\}$.

Such a notion of concreteness becomes useful when abstraction is applied to analyze problems, as one can focus on the atoms deemed to be important. For this, the user should have an idea of the atoms that matter for determining a valid solution. E.g. for planning problems, this notion can help in focusing on the actions and directly affected objects, which serve to describe a solution. One then obtains abstract answer sets that have concrete truth assignments of these atoms, while the auxiliary atoms and their concrete truth assignments become irrelevant.

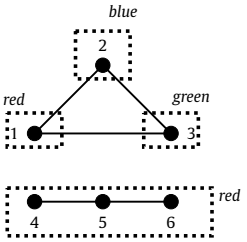


Fig. 5. Concreteness w.r.t. projection over nodes $\{1, 2, 3\}$.

3.4. Properties of domain abstraction

We now consider some basic semantic properties of our formulation of program abstraction. (Non-)existing spurious answer sets allow us to infer properties of the original program.

Proposition 3.3. *For any program* Π*,*

(i) $AS(\Pi^{m_{id}}) = \{I \cup \mathcal{T}_{m_{id}} \mid I \in AS(\Pi)\}$ *for the identity mapping* $m_{id} = \{\{x\} \mapsto x \mid x \in D\}$.
(ii) $AS(\Pi^m) = \emptyset$ *implies that* $AS(\Pi) = \emptyset$.
(iii) $AS(\Pi) = \emptyset$ *iff some* Π^m *has only spurious answer sets.*

Proof. (i) Having the identity mapping *id* causes to only have singleton clusters in the abstract domain, thus resulting in only τ_{I} and τ_{II} type facts in $\mathcal{T}_{m_{id}}$. This causes for only the rules of step (a) in Definitions 3.4 and 3.11 to remain when the rules are grounded to the relation types. Hence, the same answer sets are obtained.
(ii) Corollary of Theorem 3.2.
(iii) If $AS(\Pi) = \emptyset$, then no $\hat{I} \in AS(\Pi^m)$ for any m has a concrete answer set in Π; thus, all abstract answer sets of Π^m are spurious. Now assume the latter holds but $AS(\Pi) \neq \emptyset$. Then Π has some answer set I, and by Theorem 3.2, $m(I) \cup \mathcal{T}_m \in AS(\Pi^m)$, which would contradict that Π^m has only spurious answer sets. □

The abstract program is built by a syntactic transformation. The abstraction over the domain can also be done incrementally which in the end amounts to the overall abstraction. To establish this formally, recall from Definition 3.3 that $\mathcal{T}_m$ contains all type atoms $\tau_t^\circ(\hat{d}_1, \hat{d}_2)$ for a mapping m; we let $\mathcal{T}_{m_i,m_j} = \mathcal{T}_{m_i} \circ \mathcal{T}_{m_j}$ where $m_i \circ m_j$ is the composition of m_i and m_j.

Lemma 3.4. *For any program* Π *and mappings* m, m_1, m_2 *such that* $m_2(m_1(D)) = m(D)$*, we have* $grd_{\mathcal{T}_{m_2,m_1}}((\Pi^{m_1})^{m_2}) = grd_{\mathcal{T}_m}(\Pi^m)$*, where* $grd_{\mathcal{T}_*}$ *denotes the grounding of the program to the relation type facts* $\mathcal{T}_*$*.*

For proving Lemma 3.4, we use the following result.

Lemma 3.5. *For a relation* $d_1 \circ d_2$ *and mappings* m, m_1, m_2 *such that* $m_2(m_1(D)) = m(D)$*, we have* $\mathcal{T}^\circ_{m_2,m_1} = \mathcal{T}^\circ_m$*, where* $\mathcal{T}^\circ_*$ *denotes the set of type atoms only related with the relation* $\circ$*.*

Proof. The relation type computation $\mathcal{T}^\circ_{m_1}$ is done for $m_1(d_1) \circ m_1(d_2)$ and then the relation type computation $\mathcal{T}^\circ_{m_2,m_1}$ for $m_2(m_1(d_1)) \circ m_2(m_1(d_2)) = m(d_1) \circ m(d_2)$, resulting in the same relation type facts of $\mathcal{T}^\circ_m$. □

Proof of Lemma 3.4. From the rules of Π^{m_1}, the rules for $(\Pi^{m_1})^{m_2}$ will be constructed according to Definitions 3.4 and 3.11. Consider a rule r with body $B^{std}(r), t_1 \circ t_2$ in Π. The set $r^{m_1} \in \Pi^{m_1}$ contains rules with body $m_1(B^{std}(r))$, $\hat{t}_1 \circ \hat{t}_2$, $\tau_{\text{i}}^\circ(\hat{t}_1, \hat{t}_2)$ where $\hat{t}_k = m_1(t_k)$ if t_k is a constant; $\hat{t}_k = t_k$ otherwise.

For the set r^{m_1} of rules, a new set $(r^{m_1})^{m_2}$ will be constructed. Let $r' \in r^{m_1}$, its body will be abstracted to

$$m_2(B(r')), rel(\hat{\hat{t}}_1, \hat{\hat{t}}_2), \tau_{\text{j}}^\circ(\hat{\hat{t}}_1, \hat{\hat{t}}_2) \tag{51}$$

where $m_2(B(r')) = m_2(m_1(B^{std}(r))), m_2(\tau_{\text{i}}^\circ(\hat{t}_1, \hat{t}_2))$ and $\hat{\hat{t}}_j = m_2(m_1(t_k))$ if t_k is a constant; $\hat{\hat{t}}_k = t_k$ otherwise. Since $m_2(\tau_{\text{i}}^\circ(\hat{t}_1, \hat{t}_2)) = \tau_{\text{i}}^\circ(m_2(\hat{t}_1), m_2(\hat{t}_2)) = \tau_{\text{i}}^\circ(\hat{\hat{t}}_1, \hat{\hat{t}}_2)$, (51) will take the form

$$m(B^{std}(r)), \tau_{\text{i}}^\circ(\hat{\hat{t}}_1, \hat{\hat{t}}_2), rel(\hat{\hat{t}}_1, \hat{\hat{t}}_2), \tau_{\text{j}}^\circ(\hat{\hat{t}}_1, \hat{\hat{t}}_2).$$

where $\hat{\hat{t}}_k = m(t_k)$ if t_k is a constant; $\hat{\hat{t}}_k = t_k$ otherwise.

$c(X) \leftarrow not\ d(X), dom(X).$

$\{c(X)\} \leftarrow not\ isSingleton(X), dom(X).$

$d(X) \leftarrow not\ c(X), dom(X).$

$\{d(X)\} \leftarrow not\ isSingleton(X), dom(X).$

$b(X,Y) \leftarrow a(X), d(Y).$

$e(k_0) \leftarrow c(k_0), a(k_1).$ $e(k_0) \leftarrow c(k_0), a(k_2).$ $e(k_1) \leftarrow c(k_1), a(k_2).$ $e(k_1) \leftarrow c(k_1), a(k_0).$ $e(k_2) \leftarrow c(k_2), a(k_0).$

$e(k_2) \leftarrow c(k_2), a(k_1).$ $e(k_2) \leftarrow c(k_2), a(k_2).$ $\{e(k_0)\} \leftarrow c(k_0), a(k_0).$ $\{e(k_1)\} \leftarrow c(k_1), a(k_1).$ $\bot \leftarrow b(k_2,k_2), e(k_2).$

(a) $grd_{\mathcal{T}_{m_1}}(\Pi^{m_1})$

$c(X) \leftarrow not\ d(X), dom(X).$

$\{c(X)\} \leftarrow not\ isSingleton(X), dom(X).$

$d(X) \leftarrow not\ c(X), dom(X).$

$\{d(X)\} \leftarrow not\ isSingleton(X), dom(X).$

$b(X,Y) \leftarrow a(X), d(Y).$

$\{e(a_0)\} \leftarrow c(a_0), a(a_0).$

(b) $grd_{\mathcal{T}_{m_2,m_1}}((\Pi^{m_1})^{m_2})$

Fig. 6. Abstract programs of Example 3.28 with $m_1 = \{\{1,2\}\mapsto k_0, \{3,4\}\mapsto k_1, \{5\}\mapsto k_2\}$ and $m_2 = \{\{k_0,k_1,k_2\}\mapsto a_0\}$.

The rules in $(r^{m_1})^{m_2}$ where types of the relation differ, i.e., $i \neq j$ for $\tau_i^\circ(\hat{\hat{t}}_1, \hat{\hat{t}}_2), \tau_j^\circ(\hat{\hat{t}}_1, \hat{\hat{t}}_2)$, are insignificant as the atoms cannot both hold true in $\mathcal{T}_{m_2,m_1}$, i.e., they do not appear in $grd_{\mathcal{T}_{m_2,m_1}}((r^{m_1})^{m_2})$. As for the remaining rules in $(r^{m_1})^{m_2}$, they correspond to the rules in r^m. Thus, by Lemma 3.5 and $\{m_2(m_1(p(\vec{c}))). \mid p(\vec{c}). \in \Pi\} = \{m(p(\vec{c})). \mid p(\vec{c}). \in \Pi\}$, we obtain $grd_{\mathcal{T}_{m_2,m_1}}((\Pi^{m_1})^{m_2}) = grd_{\mathcal{T}_m}(\Pi^m)$. □

Example 3.28 *(Example 3.1 ctd).* Applying first the mapping $m_1 = \{\{1,2\}\mapsto k_0, \{3,4\}\mapsto k_1, \{5\}\mapsto k_2\}$ and then the mapping $m_2 = \{\{k_0,k_1,k_2\}\mapsto a_0\}$ yields the mapping $m = \{\{1,2,3,4,5\}\mapsto a_0\}$. Fig. 6 shows the constructed abstract programs. Notice that the program in Fig. 6b is the same as the non-ground program in Example 3.19 updated for the mapping m, i.e., a_0 is replaced with k, when it is grounded to $\mathcal{T}_{m_2,m_1}$.

An easy induction argument shows then the possibility of doing abstraction sequentially, by having abstract mappings defined over previously abstracted domains.

Proposition 3.6. *For any program* Π *and mappings* $m, m_1, \ldots, m_n$ *such that* $m_n(\ldots(m_1(D))) = m(D)$*, we have* $grd_{\mathcal{T}_m}(\Pi^m) = grd_{\mathcal{T}_{m_n,\ldots,m_1}}(((\Pi^{m_1})^{\cdots})^{m_n})$.

In Section 5.2 below, we demonstrate further uses of having a hierarchy of abstractions.

We remark that general properties of spurious answer sets from over-approximation apply to domain abstraction as an instance of it. Examples of such properties, mentioned for omission abstraction in [110,111], are *non-reoccurrence after elimination*, i.e., if a spurious answer $\hat{I}$ set of a program Π^m has no corresponding (cannot be mapped to some) answer set $\hat{I}'$ in a refinement m' of m, then no refinement m'' of m' will have an answer set $\hat{I}''$ corresponding to $\hat{I}$ either, and *convexity*, i.e., if on the contrary $\hat{I}$ has some corresponding answer set $\hat{I}'$ under m', then every refinement m'' in between m' and m admits a spurious answer set $\hat{I}''$ of $\Pi^{m''}$ that corresponds with $\hat{I}$.

3.4.1. Abstraction over sorts

Applications of ASP usually contain *sorts* that form subdomains of the Herbrand universe. For example, in graph coloring there are sorts for nodes and colors. We define an abstraction over a sort as follows.

Definition 3.14. An abstraction is limited to a sort $D_i \subseteq D$, if all elements $x \in D \setminus D_i$ form singleton clusters $\{x\}\mapsto x$.

Example 3.29. In graph coloring, we have sorts *node* and *color* in the domain $\{1,\ldots,6, red, green, blue\}$ for the instance in Fig. 1a. An abstraction mapping m limited to the sort *node* means $m(x) = \{x\}$ for $x \in \{red, blue, green\}$.

In order to obtain much coarser abstractions, applying abstraction over multiple sorts is also possible, given that the individual sorts fulfill the following property.

Definition 3.15 *(sort independence).* For a program Π and domain D, subdomains $D_1, \ldots, D_n \subseteq D$ are *independent* if $D_i \cap D_j = \emptyset$ for all $i \neq j$.

For independent sorts, abstractions can be composed.

(a) A concrete answer set (b) A spurious answer set

Fig. 7. Abstraction over the set of nodes and the set of colors.

Proposition 3.7. *For every domain mappings m_1 and m_2 over independent subdomains D_1 and D_2, it holds that* $grd_{\mathcal{T}_{m_1,m_2}}((\Pi^{m_2})^{m_1}) = grd_{\mathcal{T}_{m_2,m_1}}((\Pi^{m_1})^{m_2})$.

Proof. The mapping $m_i : D \mapsto \widehat{D}$ is of form $\{\{x\} \mapsto x \mid x \in D \setminus D_i\} \cup m_{D_i}$, $i \in \{1,2\}$, where m_{D_i} describes the mapping over D_i to the abstract domain $\widehat{D}_i$. We know that $m_i(D \setminus D_i) = D \setminus D_i$, and since D_1 and D_2 are independent, we have $D_1 \subseteq m_2(D)$ and $D_2 \subseteq m_1(D)$. Consequently, we can apply the mappings independently from each other as $m_2(m_1(D)) = m_1(m_2(D))$ to achieve an abstract domain $\widehat{\widehat{D}} = (D \setminus (D_1 \cup D_2)) \cup \widehat{D}_1 \cup \widehat{D}_2$. Another mapping m can then be defined to map D to $\widehat{\widehat{D}}$. By Lemma 3.4 we get the result. □

3.4.2. Cartesian abstraction

Given domain mappings $m_1, \ldots, m_n$ limited to subdomains $D_1, \ldots, D_n$, respectively, a *cartesian abstraction* of the mappings corresponds to the abstract domain $m(D_1) \times \cdots \times m(D_n)$. Assuming that the subdomains $D_1, \ldots, D_n$ are independent, Definition 3.4 can be altered to be applied over a rule of the form

$$r : \alpha \leftarrow B^{std}(r), t_1 \circ^{D_1} t_2, \ldots, t_1 \circ^{D_n} t_2$$

by considering all possible combinations of $\tau_j^{\circ^{D_i}}(m(t_1), m(t_2))$, $j = 1, \ldots, n$. Alternatively, we can define cartesian abstraction by applying abstraction over each subdomain one step at a time, by extending Proposition 3.7 to multiple sorts.

Proposition 3.8. *For domain mappings $m_1, \ldots, m_n$ over independent domains $D_1, \ldots, D_n$, it holds that* $\Pi^{m_1 \times \ldots \times m_n} = ((\Pi^{m_{\pi(1)}})^{\cdots})^{m_{\pi(n)}}$ *where π is any permutation of $\{1, \ldots, n\}$.*

Example 3.30 *(Example 2.4 ctd).* In the graph coloring instance of Fig. 1a, consider the mappings $m_n = \{\{4,5,6\} \mapsto \hat{4}\}$ and $m_c = \{\{red\} \mapsto \hat{r}, \{green, blue\} \mapsto \hat{gb}\}$ over the sorts nodes and colors, respectively. The abstract program $(\Pi^{m_n})^{m_c}$ has the concrete answer set

$$\{chosenColor(1, \hat{r}), chosenColor(2, \hat{gb}), chosenColor(3, \hat{gb}), chosenColor(\hat{4}, \hat{r})\}$$

(shown in Fig. 7a) that chooses the color cluster $\hat{gb}$ for nodes 2 and 3, which matches the intuition of coloring the neighbor nodes of node 1 to some color different than its own color.

Notably, $(\Pi^{m_n})^{m_c}$ also has the spurious answer set (shown in Fig. 7b)

$$\{chosenColor(1, \hat{gb}), chosenColor(2, \hat{gb}), chosenColor(3, \hat{gb}), chosenColor(\hat{4}, \hat{r})\}$$

due to the guesses introduced for the uncertainty.

In Section 7.2 we demonstrate further uses of such an multi-step abstraction over the subdomains.

3.5. Computational complexity

In this section, we turn to the computational complexity of reasoning tasks that are associated with program abstraction. We build on the complexity results in [33,39], which cover the basic reasoning tasks for arbitrary non-ground programs and for non-ground programs with bounded predicate arities, i.e., the maximum arity of a predicate occurring in the program is bounded by a constant.

Lemma 3.9. *Given an arbitrary non-ground program Π, a mapping m, and an abstract interpretation $\hat{I}$, checking whether $\hat{I} \in AS(\Pi^m)$ holds is feasible in Δ_2^p.*

Intuitively, this holds because we can nondeterministically generate each rule r in Π^m in polynomial time, and if I is not a model of the reduct $(\Pi^m)^I$ also an instance of r witnessing this fact. The minimality of I for $(\Pi^m)^I$ can be shown by a polynomial-size proof tree that can be guessed and checked.

Armed with this lemma, we consider the problem of identifying concrete abstract answer sets.

Theorem 3.10. *Given a program Π, a domain mapping m, and an abstract interpretation $\hat{I}$, deciding whether $\hat{I}$ is a concrete abstract answer set of Π^m is* NEXP*-complete in general and Σ_2^p-complete for bounded predicate arities. Furthermore, the complexity remains unchanged if $\hat{I} \in AS(\Pi^m)$ is asserted.*

That is, the worst case complexity is the one of answer set existence for non-ground programs [33,39]; the two problems can be reduced to each other in polynomial time. Intuitively, in general an abstract atom in $\hat{I}$ may be mapped back to exponentially many atoms in an answer set I of the original program Π that witnesses the concreteness of $\hat{I}$; such an I can be guessed and checked in nondeterministic exponential time. Accordingly, the complexity drops to Σ_2^p if the domain size $|D|$ is polynomial in the abstracted domain size $|\widehat{D}|$ and interpretations are represented as bitmaps (as customary); e.g., it drops if each abstract cluster is small (and multiple clusters exist). Under bounded predicate arities, each abstract atom maps back to polynomially many original atoms, such that the guess I has polynomial size and checking I can be done with an NP oracle in polynomial time (cf. Lemma 3.9). The matching lower bounds are shown by reductions from deciding whether a given non-ground program has some answer set.

As an immediate consequence of Theorem 3.10, we obtain the following result for spuriousness checking.

Corollary 3.11. *Given a program Π, a domain mapping m, and an abstract interpretation $\hat{I}$, deciding whether $\hat{I}$ is a spurious abstract answer set of Π^m is* coNEXP*-complete in general and Π_2^p-complete for bounded predicate arities. Furthermore, the complexity remains unchanged if $\hat{I} \in AS(\Pi^m)$ is asserted.*

Next we consider deciding whether the abstract program has some spurious answer set. This problem turns out to have higher complexity.

Theorem 3.12. *Given a program Π and a domain mapping m, deciding whether some $\hat{I} \in AS(\Pi^m)$ exists that is spurious is* $\mathsf{NEXP}^{\mathsf{NP}}$*-complete in general and Σ_3^p-complete for programs with bounded predicate arities.*

Intuitively, compared to the previous problem we first must make a guess for $\hat{I}$ such that it is an abstract answer set of Π^m but not concrete; the size of $\hat{I}$ may be exponential in the input of the problem, and relative to this testing concreteness is feasible in nondeterministic polynomial time, i.e., with an NP oracle. The matching hardness is shown by reductions from evaluating second-order logic formulas of a suitable form over finite relational structures.

Faithful abstraction. An abstract program that does not have a spurious answer set is a faithful abstraction of the original program.

Example 3.31 *(Example 2.4 ctd).* In the graph coloring instance of Fig. 1a, the mapping $m = \{\{4, 5, 6\} \mapsto \hat{4}\}$, which maps nodes $1, 2, 3$ to singleton clusters, yields an abstract program Π^m that has 42 answer sets, which are the combinations of 6 possible correct colorings of the nodes $1 - 3$ with 7 possible colorings $\{\{red\}, \{blue\}, \{green\}, \{red, blue\}, \{red, green\}, \{green, blue\}, \{red, green, blue\}\}$ of the node cluster $\hat{4}$, thus resulting in a faithful abstraction.

Ideally, we have faithfulness, but this is hard to achieve in general. From Theorem 3.12, we immediately obtain:

Corollary 3.13. *Given a program Π and a domain mapping m, deciding whether Π^m is faithful is* $\mathsf{coNEXP}^{\mathsf{NP}}$*-complete in general and Π_3^p-complete for bounded predicate arities.*

4. Refinement by debugging non-ground spuriousness

Over-approximation of an answer set program unavoidably introduces spurious answer sets. Once a spurious abstract answer set is encountered, one can either continue searching for a concrete abstract answer set, or *refine* the abstraction to reach one where less spurious answer sets occur.

Definition 4.1. Given a domain mapping $m : D \to \widehat{D}'$, a mapping $m' : D \to \widehat{D}''$ is a *refinement* of m if for all $x \in D$, $m'^{-1}(m'(x)) \subseteq m^{-1}(m(x))$.

That is, refinement is on dividing the abstract clusters to a finer grained domain.

Example 4.1 *(Example 3.21 ctd).* The mapping $m' = \{\{1\}\mapsto k_1, \{2,3\}\mapsto k_2, \{4,5\}\mapsto k_3\}$ is a refinement of the mapping $m = \{\{1,\ldots,5\}\mapsto k\}$. Furthermore, $\Pi^{m'}$ has no answer set $\hat{I}'$ such that $m(m'^{-1}(\hat{I}')) = \hat{I}_2 = \{a(k), c(k)\}$; hence the spurious answer set $\hat{I}_2$ of Π^m is eliminated.

In the CEGAR methodology [25], the decision in a refinement step depends on the correctness checking of the spurious abstract solution, through which the problematic part of the abstraction is detected. Inspired by this, we develop an alternative for checking the correctness of abstract answer sets that can be used to determine how the refinement should be made.

Correctness checking using constraints. That an abstract answer set $\hat{I}$ is spurious means the original program Π has no answer set matching $\hat{I}$. In other words, querying Π for a match to an abstract answer set $\hat{I}$ would return no result exactly if $\hat{I}$ is spurious.

Definition 4.2 *(query of an answer set).* Given an abstract answer set $\hat{I}$ and a mapping m, a *query* $Q^m_{\hat{I}}$ for an answer set that matches $\hat{I}$ is described by the following constraints.

$$\bot \leftarrow \{\alpha \mid m(\alpha) = \hat{\alpha}\} \leq 0. \qquad \hat{\alpha} \in \hat{I} \setminus \mathcal{T}_m, \tag{52}$$

$$\bot \leftarrow \alpha. \qquad \hat{\alpha} \notin \hat{I} \setminus \mathcal{T}_m, m(\alpha) = \hat{\alpha}. \tag{53}$$

Here (52) ensures that a witnessing answer set I of Π (i.e., $m(I) = \hat{I}$) contains for every non-τ_t abstract atom in $\hat{I}$ some atom that is mapped to it, while (53) ensures that no atom in I is mapped to an abstract atom not in $\hat{I}$. The following is then easy to establish.

Proposition 4.1. *Suppose m is a domain abstraction mapping for a program Π, then an abstract answer set $\hat{I} \in AS(\Pi^m)$ is spurious iff $\Pi \cup Q^m_{\hat{I}}$ is unsatisfiable.*

Proof. As $\hat{I}$ is spurious, there exists no $I \in AS(\Pi)$ such that $m(I) = \hat{I} \setminus \mathcal{T}_m$, i.e., there is no match of an original answer set I for $\hat{I}$ where the atoms in I can be mapped to the abstract atoms contained in $\hat{I} \setminus \mathcal{T}_m$ and the atoms not in I can be mapped to the abstract atoms not contained in $\hat{I} \setminus \mathcal{T}_m$. $Q_{\hat{I}}$ enforces such a match, thus returns unsatisfiability.

Having no match for $\hat{I}$ means that no original answer set can be mapped to it, thus $\hat{I}$ is spurious. □

Correctness checking with debugging. We will employ an ASP debugging approach to debug the inconsistency of the original program Π caused by checking a spurious answer set $\hat{I}$, referred to as *inconsistency of* Π *w.r.t.* $\hat{I}$, in order to get hints for refining the abstraction. Different from a usual ASP program debugging approach, we need to shift the focus from "debugging the original program" to "debugging the inconsistency caused by the spurious answer set". Unfortunately an immediate application of the available ASP debugging tools is not possible. For our purposes, we make use of the meta-level debugging language in [16], which is based on a tagging technique that allows one to control the building of answer sets and to manipulate the program evaluation.

The meta-program constructed by `spock` [16] introduces *tags* to control answer set building. Given a ground program Π that is viewed as program over a propositional alphabet (i.e., ground atoms are propositional atoms) and a set $\mathcal{N}$ of names for all rules in Π, it creates an enriched alphabet by adding propositional atoms such as $ap(n_r), bl(n_r), ok(n_r), ko(n_r)$ where $n_r \in \mathcal{N}$ for each $r \in \Pi$. The atoms $ap(n_r), bl(n_r)$ express whether a rule r is applicable or blocked, respectively, while $ok(n_r), ko(n_r)$ are used for manipulating the application of r.

For domain abstraction, debugging the non-ground program has its own difficulties. The approach in [16] is on the propositional level, thus cannot be immediately applied. Also debugging non-ground programs is not as straightforward as in the propositional case, as there is the additional need to debug the checking for an original answer set that can be mapped to the given abstract answer set. However, non-ground program debugging approaches such as [36,93] are not easily adjustable due to the need for shifting the focus towards debugging the correctness check.

Using available debuggers. Debugging non-ground ASP programs through a meta-programming [55] approach has been studied by [93], with the drawback of considering all possible explanations why a given interpretation I is not an answer set of the program Π. For the given input I, in order to prove that I is not an answer set of Π, the debugging considers many possible guesses of variable assignments that matches I with a faulty behavior. In our case, the input I is an abstract answer set stating that there should be some original answer set I' of Π such that each atom in I' can be mapped to some abstract atom $\hat{\alpha}$ in I. This adds a guess of some original atom that could be mapped to $\hat{\alpha}$. However, as the debugging aims at showing that I is not an answer set of Π, when this additional guessing comes into play, it guesses original atoms to create some faulty behavior for I even if these atoms do not occur in an original answer set. Thus, an immediate application of the meta-programming approach is infeasible.

In order to use the available non-ground debugging tools off-the-shelf, one possibility is to first guess all possible combinations of the original atoms to match the abstract answer set $\hat{I}$, and then separately debug each of them. If $\hat{I}$ is in fact

spurious, this will be caught as each possible guess would return some inconsistency. If $\hat{I}$ is concrete, then at some point some guess will correspond to an original answer set, with no inconsistency. However, this approach is too cumbersome, as there can be many possible concrete guesses for an abstract I and checking each of them one by one until a concrete one is found (if it exists) is highly inefficient.

Our approach to debugging. As existing non-ground debugging tools are not readily applicable, we debug the unsatisfiability of $\Pi \cup Q_{\hat{I}}^m$ for a spurious abstract answer set $\hat{I}$ following the debugging approach based on [16] from above. In previous work on domain abstraction [113], we considered a simplified debugging approach inspired from the *ko* atoms of [16], which is based on detecting the rules that must be deactivated in order to keep the satisfiability while checking the concreteness of an abstract answer set $\hat{I}$, in case it is spurious. As the naive debugging cannot address all debugging cases, in this work we show an extension of the refinement method by lifting the `spock` [16] debugging approach to the non-ground case, confining to a class of programs that subsumes tight programs.

When demonstrating the different debugging approaches, we use a non-ground version of $Q_{\hat{I}}^m$.

Definition 4.3 *(non-ground query).* Given an abstract answer set $\hat{I}$ and a mapping m expressed as a set of facts of form $m(x, \widehat{d})$ (where $m(x) = \widehat{d}$), a *(non-ground) query* for an answer set that matches $\hat{I}$ is described as follows:

$$\bot \leftarrow in_p(\hat{X}_1, \ldots, \hat{X}_k), \{p(X_1, ..., X_k) : m(X_1, \hat{X}_1), \ldots, m(X_k, \hat{X}_k)\} \leq 0. \tag{54}$$

$$\bot \leftarrow p(X_1, ..., X_k), \mathit{not}\ in_p(\hat{X}_1, \ldots, \hat{X}_k), m(X_1, \hat{X}_1), \ldots, m(X_k, \hat{X}_k). \tag{55}$$

$$in_p(\hat{d}_1, \ldots, \hat{d}_k). \qquad \text{for all } p(\hat{d}_1, ..., \hat{d}_k) \in \hat{I} \setminus \mathcal{T}_m, \qquad \text{where } p \neq dom \text{ is a non-relational predicate.} \tag{56}$$

Example 4.2 *(Example 3.1 ctd).* For the program Π and the mapping $m = \{\{1, 2, 3, 4, 5\} \mapsto k\}$ given as facts $m(1, k)$, $m(2, k), m(3, k), m(4, k), m(5, k)$, the abstract program Π^m has an answer set $\hat{I} = \{a(k), c(k)\}$. The query $Q_{\hat{I}}^m$ is

$$\bot \leftarrow \mathit{not}\ in_d(A_1), d(X_1), m(X_1, A_1). \qquad \bot \leftarrow in_d(A_1), \{d(X_1) : m(X_1, A_1)\} \leq 0.$$

$$\bot \leftarrow \mathit{not}\ in_c(A_1), c(X_1), m(X_1, A_1). \qquad \bot \leftarrow in_c(A_1), \{c(X_1) : m(X_1, A_1)\} \leq 0.$$

$$\bot \leftarrow \mathit{not}\ in_a(A_1), a(X_1), m(X_1, A_1). \qquad \bot \leftarrow in_a(A_1), \{a(X_1) : m(X_1, A_1)\} \leq 0.$$

$$\bot \leftarrow \mathit{not}\ in_e(A_1), e(X_1), m(X_1, A_1). \qquad \bot \leftarrow in_e(A_1), \{e(X_1) : m(X_1, A_1)\} \leq 0.$$

$$\bot \leftarrow \mathit{not}\ in_b(A_1, A_2), b(X_1, X_2), m(X_1, A_1), m(X_2, A_2).$$

$$\bot \leftarrow in_b(A_1, A_2), \{b(X_1, X_2) : m(X_1, A_1), m(X_2, A_2)\} \leq 0.$$

$$in_a(k). \qquad in_c(k).$$

4.1. Non-ground debugging using tagging

We extend the refinement method described in [113] by lifting the "tagging" approach of `spock` [16] to the non-ground case, confining to a class of programs that makes it possible to avoid unfounded loop checking when debugging the spuriousness query. Given Π, we construct the meta program $\mathcal{T}_{meta}[\Pi]$ similar to `spock` [16], but with an extension of having arguments in the ap_{n_r}, bl_{n_r} atoms to have information for which constants the rules are applicable and blocked.

Definition 4.4. Given a non-ground program Π, the program $\mathcal{T}_{meta}[\Pi]$ over the vocabulary $\mathcal{V}_{meta}$ that enriches the vocabulary $\mathcal{V}$ of Π with predicates ap_{n_r}, bl_{n_r}, ap_{n_r}, ko_{n_r} for each $n_r \in \mathcal{N}$, consists of the following rules for $r \in \Pi$ with $\{c_1, \ldots, c_n\}$ denoting the set of terms that are arguments of $H(r)$, and $\{d_1, \ldots, d_{n'}\}$ denoting the set of terms that are arguments of literals in $B(r)$:

$$\text{If } B(r) = \emptyset: \qquad r$$

$$\text{If } H(r) \neq \bot \wedge n > 0: \quad \begin{cases} H(r) \leftarrow ap_{n_r}(c_1, \ldots, c_n), \mathit{not}\ ko_{n_r}. \\ ap_{n_r}(c_1, \ldots, c_n) \leftarrow B(r). \\ bl_{n_r}(c_1, \ldots, c_n) \leftarrow \mathit{not}\ ap_{n_r}(c_1, \ldots, c_n). \end{cases}$$

$$\text{If } H(r) = \bot \vee n = 0: \quad \begin{cases} H(r) \leftarrow ap_{n_r}(d_1, \ldots, d_{n'}), \mathit{not}\ ko_{n_r}. \\ ap_{n_r}(d_1, \ldots, d_{n'}) \leftarrow B(r). \\ ap_{n_r} \leftarrow ap_{n_r}(d_1, \ldots, d_{n'}). \\ bl_{n_r} \leftarrow \mathit{not}\ ap_{n_r}. \end{cases}$$

In case the head of rule r is $\bot$ or does not contain arguments in the atom, we use the arguments from the body to know whether r is applicable.

We have abnormality atoms to indicate the actions that are required to avoid the inconsistency:

- $ab\_deact_{n_r}$ signals that r was applicable under some interpretation, but had to be deactivated;
- similarly for $ab\_deactCons_{n_r}$ which only talks about the constraints; and
- $ab\_act_p(c_1,\ldots,c_n)$ for $\alpha = p(c_1,\ldots,c_n)$ says that atom α must be made true while no rule deriving α was applicable (i.e., α is *unsupported*[4]).

Definition 4.5. Given a non-ground program Π with vocabulary $\mathcal{V}$, the following additional meta-programs are constructed over the vocabulary $\mathcal{V}_{debug}$ which enriches $\mathcal{V}_{meta}$ with abnormality predicates $ab\_deact_{n_r}$, $ab\_deactCons_{n_r}$, and ab_act_p; and with $c_1,\ldots,c_n,d_1,\ldots,d_{n'}$ as in Definition 4.4, as follows:

1. Rule Deactivation: $\mathcal{T}_{deact}[\Pi]$: for all $r \in \Pi$ with $B(r) \neq \emptyset$ and $H(r) \neq \bot$:

$$ko_{n_r}.$$
$$\{H(r)\} \leftarrow ap_{n_r}(c_1,\ldots,c_n).$$
$$ab\_deact_{n_r}(c_1,\ldots,c_n) \leftarrow ap_{n_r}(c_1,\ldots,c_n), not\ H(r).$$

2. Constraint Deactivation: $\mathcal{T}_{deactCons}[\Pi]$: for all $r \in \Pi$ with $H(r) = \bot$:

$$\{ko_{n_r}\}.$$
$$ab\_deactCons_{n_r}(d_1,\ldots,d_{n'}) \leftarrow ap_{n_r}(d_1,\ldots,d_{n'}), ko_{n_r}.$$

3. Rule Head Activation: $\mathcal{T}_{act}[\Pi,\mathcal{V}]$: for all rule heads $\alpha = p(c_1,\ldots,c_n)$ in Π with $\alpha \in \mathcal{V}$ and $pdef(p,\Pi) = \{r_1,\ldots,r_k\}$ and $k \geq 1$:

$$\{\alpha\} \leftarrow bl_{n_{r_1}}(c_1,\ldots,c_n),\ldots,bl_{n_{r_k}}(c_1,\ldots,c_n).$$
$$ab\_act_p(c_1,\ldots,c_n) \leftarrow \alpha, bl_{n_{r_1}}(c_1,\ldots,c_n),\ldots,bl_{n_{r_k}}(c_1,\ldots,c_n).$$

The arguments of *ab_deact* only contain the ones from the head of the rule. This is a representation choice, to avoid dealing with many variables involved in the body while only few of them are used in the head of the rule. For the definition of *ab_deactCons* however, the variables of the body must be used. Having a different representation for the deactivation of the constraints will allow to steer the debugging towards the constraints by assigning different costs for their occurrence when computing the answer sets with the smallest number of *ab* atoms.

Definition 4.6 *(debugging program Π_{debug}).* For a program Π over vocabulary $\mathcal{V}$, we let the program Π_{debug} over $\mathcal{V}_{debug}$ be defined by

$$\Pi_{debug} = \mathcal{T}_{meta}[\Pi] \cup \mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi,\mathcal{V}].$$

We use Π_{debug} for checking the correctness of an abstract answer set and then deciding on the refinement. Adding weak constraints over the abnormality atoms yields an answer set with fewest *ab* atoms. We here use *weak constraints* [74] of the form

$$\bot :\sim \alpha_1,\ldots,\alpha_m, not\ \alpha_{m+1},\ldots,not\ \alpha_n.\ [w,t_1,\ldots,t_k]$$

where w (the *weight*) is a positive integer constant or variable, and $t_1,\ldots,t_k$ are terms from $\alpha_1,\ldots,\alpha_n$. For each answer set, the set of all tuples $(w,t_1,\ldots,t_k)$ of violated weak constraints is computed, and the sum of the first components of this set is assigned to the answer set as a cost. Among all answer sets, those whose cost is smallest are chosen as *optimal answer sets*. Using weak constraints is a convenient way of performing optimizations.

Example 4.3 *(Example 4.2 ctd).* The program Π_{debug} with additional weak constraints on abnormality atoms is shown in Fig. 8. The minimal answer set of $\Pi_{debug} \cup Q_I^m$ is then

$$\{ab\_deact_{r4}(1), ab\_deact_{r4}(2),\ldots,ab\_deact_{r4}(5)\}.$$

This debugging approach is also able to handle the shortcomings of the naive approach [113], as $\mathcal{T}_{act}[\Pi,\mathcal{V}]$ is used to activate original atoms if it is necessary for achieving satisfiability for $\Pi_{debug} \cup Q_I^m$.

[4] An atom α is *unsupported* by an interpretation I if for each $r \in def(\alpha,\Pi)$, $B^+(r) \not\subseteq I$ or $B^-(r) \cap I \neq \emptyset$ [120].

$$c(X) \leftarrow ap_{r1}(X), not\ ko_{r1}. \qquad e(X) \leftarrow ap_{r4}(X), not\ ko_{r4}.$$
$$ap_{r1}(X) \leftarrow not\ d(X). \qquad ap_{r4}(X) \leftarrow c(X), a(Y), X \neq Y.$$
$$bl_{r1}(X) \leftarrow not\ ap_{r1}(X). \qquad bl_{r4}(X) \leftarrow not\ ap_{r4}(X).$$

$$d(X) \leftarrow ap_{r2}(X), not\ ko_{r2}. \qquad \bot \leftarrow ap_{r5}(X,Y), not\ ko_{r5}.$$
$$ap_{r2}(X) \leftarrow not\ c(X). \qquad ap_{r5}(X,Y) \leftarrow b(X,Y), e(X).$$
$$bl_{r2}(X) \leftarrow not\ ap_{r2}(X). \qquad ap_{r5} \leftarrow ap_{r5}(X,Y).$$
$$bl_{r5} \leftarrow not\ ap_{r5}.$$

$$b(X,Y) \leftarrow ap_{r3}(X,Y), not\ ko_{r3}. \qquad a(1).\ a(3).$$
$$ap_{r3}(X,Y) \leftarrow a(X), d(Y).$$
$$bl_{r3}(X,Y) \leftarrow not\ ap_{r3}(X,Y).$$

$$ko_{r1}. \qquad \{ko_{r5}\}.$$
$$\{c(X)\} \leftarrow ap_{r1}(X). \qquad ab\_deactCons_{r5}(X,Y) \leftarrow ko_{r5}, ap_{r5}(X,Y).$$
$$ab\_deact_{r1}(X) \leftarrow ap_{r1}(X), not\ c(X). \qquad \bot :\sim ab\_deactCons_{r5}(X,Y).\ [1, r5, X, Y]$$
$$\bot :\sim ab\_deact_{r1}(X).\ [1, r1, X]$$

$$\{c(X)\} \leftarrow bl_{r1}(X).$$
$$ab\_act_c(X) \leftarrow c(X), bl_{r1}(X).$$
$$\bot :\sim ab\_act_c(X).\ [1, X]$$

$$ko_{r2}.$$
$$\{d(X)\} \leftarrow ap_{r2}(X).$$
$$ab\_deact_{r2}(X) \leftarrow ap_{r2}(X), not\ d(X).$$
$$\bot :\sim ab\_deact_{r2}(X).\ [1, r2, X]$$

$$\{d(X)\} \leftarrow bl_{r2}(X).$$
$$ab\_act_d(X) \leftarrow d(X), bl_{r2}(X).$$
$$\bot :\sim ab\_act_d(X).\ [1, X]$$

$$ko_{r3}.$$
$$\{b(X,Y)\} \leftarrow ap_{r3}(X,Y).$$
$$ab\_deact_{r3}(X,Y) \leftarrow ap_{r3}(X,Y), not\ b(X,Y).$$
$$\bot :\sim ab\_deact_{r3}(X,Y).\ [1, r3, X, Y]$$

$$\{b(X,Y)\} \leftarrow bl_{r3}(X).$$
$$ab\_act_b(X,Y) \leftarrow b(X,Y), bl_{r3}(X).$$
$$\bot :\sim ab\_act_b(X,Y).\ [1, X, Y]$$

$$ko_{r4}.$$
$$\{e(X)\} \leftarrow ap_{r4}(X).$$
$$ab\_deact_{r4}(X) \leftarrow ap_{r4}(X), not\ e(X).$$
$$\bot :\sim ab\_deact_{r4}(X).\ [1, r4, X]$$

$$\{e(X)\} \leftarrow bl_{r4}(X).$$
$$ab\_act_c(X) \leftarrow e(X), bl_{r4}(X).$$
$$\bot :\sim ab\_act_e(X).\ [1, X]$$

Fig. 8. Debugging program Π_{debug} for Example 4.2 with weak constraints.

As a first property, we show that $\Pi_{debug} \cup Q_{\hat{I}}^m$ always has an answer set, i.e., no abstract answer set $\hat{I}$ is dismissed provided the program Π at hand obeys the following property. We call Π *positive-dependency founded* if no negative edge in G_Π points to a cycle in G_Π^+, i.e., atoms in positive loops are not negatively conditioned to any atom. Note that positive-dependency founded programs subsume tight programs, where G_Π^+ is acyclic.

Proposition 4.2. *Given a positive-dependency founded program Π and a mapping m, for each answer set $\hat{I} \in AS(\Pi^m)$, $\Pi_{debug} \cup Q_{\hat{I}}^m$ has an answer set.*

The next result now shows that we can use $\Pi_{debug} \cup Q_{\hat{I}}^m$ to obtain hints for the spuriousness reason of $\hat{I}$.

Proposition 4.3. *Given a positive-dependency founded program Π and a mapping m, if an answer set $\hat{I} \in AS(\Pi^m)$ is spurious, then for every answer set $S \in AS(\Pi_{debug} \cup Q_{\hat{I}}^m)$ either (i) $ab\_deact_{n_r}(c_1, \ldots, c_n) \in S$ or $ab\_deactCons_{n_r}\ (d_1, \ldots, d_{n'}) \in S$ for some $r \in \Pi$, or (ii) $ab\_act_p(c_1, \ldots, c_n) \in S$ for some $r \in grd(\Pi)$ with $H(r) = \{p(c_1, \ldots, c_n)\}$.*

Less surprisingly, for programs that are not positive-dependency founded, debugging the correctness check could result in unsatisfiability (see Appendix B.1 for an example). To avoid this, unfounded loop checking can be handled by introducing an additional abnormality atom, say ab_{loop} as in [16], and lifting it to the non-ground setting. However, this solution causes further guessing rules involved in the non-ground debugging. Also the existence of $ab_{loop}(\alpha)$ sometimes does not even indicate that a loop formula is violated and just makes the search more difficult due to considering many possibilities of the guesses. Therefore, we choose to focus only on positive-dependency founded programs and concentrate on the determination of a refinement.

The obtained debugging atoms during a correctness check give hints on which domain elements should not be involved in a cluster.

Definition 4.7 *(refinement-hint program Π_{hint}).* The refinement-hint gathering program Π_{hint} for a program Π contains the following rules, with $c_1, \ldots, c_n, d_1, \ldots, d_{n'}$ as in Definition 4.4:

- For $c_i \in arg(ab\_deact_{n_r}(c_1, \ldots, c_n))$:

$$refine(c_1, \ldots, c_n) \leftarrow ab\_deact_{n_r}(c_1, \ldots, c_n), m(c_i, a_i), not\ isSingleton(a_i).$$

Algorithm 1: *decideRefinement* with Search.

```
   Input: program Π, domain mapping m
   Output: refinement m' of m
 1 if m has non-singleton clusters then
 2     refinecosts = []; allrefs = [];
       /* compute all 1-distance refinements of m                          */
 3     refs = computeRefinements(m, 1)
 4     forall m' ∈ refs do
 5         c = getCostOfMapping(Π, m');
 6         if c = 0 then /* found a concrete abstract answer set           */
 7             return m';
 8         else
 9             allrefs.append(m');
10             refinecosts.append(c)
11     minrefs = getRefsMinCost(refinecosts, allrefs)
12     m = pickRandomRef(minrefs)
13 return m
14 ---------------------------------------------------------------------
15 def getCostOfMapping(Π, m)
16     Π^m = constructAbsProg(Π, m);
17     Π_debug = constructDebugProg(Π);
18     Pick some Î ∈ AS(Π^m)
19     Find optimum answer set I' of Π_debug ∪ Q_Î^m /* with smallest number s of ab-atoms */
20     return |I'|_ab|      /* s = |I'|_ab|                                 */
```

- For $d_i \in arg(ab\_deactCons_{n_r}(d_1, \ldots, d_{n'}))$:

$$refine(d_1, \ldots, d_{n'}) \leftarrow ab\_deactCons_{n_r}(d_1, \ldots, d_{n'}), m(d_i, a_i), not\ isSingleton(a_i).$$

- For $c_i \in arg(p(c_1, \ldots, c_n))$:

$$refine(c_1, \ldots, c_n) \leftarrow ab\_act_p(c_1, \ldots, c_n), m(c_i, a_i), not\ isSingleton(a_i).$$

From Π_{hint} we get as hints the domain elements that are mapped to abstract cluster elements and cause *ab* atoms in the debugging.

4.2. Deciding on a refinement

The introduced debugging approach finds a set of abnormality atoms in case the abstract answer set is spurious. We consider two ways of using the obtained debugging output for deciding on a refinement.

(v1) The smallest number of *ab* atoms occurring in an answer set is the cost of the corresponding mapping.
(v2) The inferred *refine* atoms are used to decide on a refinement of the abstraction.

In (v1), the cost is used for a local search among the possible refinements of an abstraction, where the one with the minimum cost is picked. Approach (v2) is closer to the CEGAR-like approach [25], where a refinement is determined from the spuriousness check. We now describe the approaches in more detail and report on a comparison in Section 6.3.

(v1) Local Refinement Search. The idea is to search among possible refinements of a mapping for deciding on a refinement. To single out close refinements, we measure the distance $dist(m, m')$ between a mapping $m{:}D \to \widehat{D}$ and a refinement $m'{:}D \to \widehat{D}'$ of it by the number of additional clusters, i.e., $dist(m, m') = |\widehat{D}'| - |\widehat{D}|$. In case $dist(m, m') = 1$, we call m' a *1-distance refinement* of m.

Example 4.4. Each mapping $m' \in \bigcup_{C \in \mathcal{C}} \{C \mapsto k_1, \{1, \ldots, 5\} \setminus C \mapsto k_2\}$, where $\mathcal{C} = \{\{1\}, \{1, 2\}, \{1, 2, 3\}, \{1, 2, 3, 4\}\}$, is a 1-distance refinement of $m = \{\{1, 2, 3, 4, 5\} \mapsto k\}$.

Algorithm 1 shows the procedure of deciding on a refinement for a given mapping m, by doing a distance-based search among all possible refinements of the mapping and picking the one with the least cost. All 1-distance refinements of m are computed, and then the cost of each of them is determined, by calling *getCostOfMapping*. This function constructs the abstract program Π^m according to the mapping and picks an abstract answer set $\hat{I}$. It then finds the answer set with smallest number s of *ab*-atoms of the program $\Pi_{debug} \cup Q_{\hat{I}}^m$ and returns s. If some refinement m' has cost 0, it is returned. Otherwise, all the refinements and their costs were collected. In Line 11 the refinements with minimum cost are gathered, and then a random pick is made over them. If the given mapping contains only singleton clusters, this means the original domain has been reached.

(v2) Abstraction Refinement Using Hints. The abstract answer set correctness checking returns *ab*-atoms that contain the domain elements involved in the debugging of the unsatisfiability. The latter can be used as hints on which part of the mapping to refine. The idea of the *refine* atoms is to get hints about which domain elements should not be involved in a cluster. Given a hint atom $\mathit{refine}(c_1, \ldots, c_n)$, we consider two actions to describe a refinement m' of m:

(1) For $\mathit{refine}(c_1, \ldots, c_n)$ and $i \in \{1, \ldots, n\}$ such that $m^{-1}(m(c_i)) > 1$, the refinement m' satisfies $m'^{-1}(m'(c_i)) = 1$.
(2) For $\mathit{refine}(c_1, \ldots, c_n)$ and $c_i \neq c_j \in \{c_1, \ldots, c_n\}$ such that $m(c_i) = m(c_j)$, the refinement m' satisfies $m'(c_i) \neq m'(c_j)$.

Applying refinement action (1) means to refine the abstraction by mapping all elements occurring in some *refine* atom to singletons, while the refinement action (2) should ensure that distinct elements occurring in the same *refine* atom are no longer mapped to the same cluster.

Example 4.5 *(Example 4.3 ctd).* The hint atoms for the minimal answer set of $\Pi_{debug} \cup Q^m_{\hat{I}}$ are $\{\mathit{refine}(1), \ldots, \mathit{refine}(5)\}$. Applying refinement action (1) means to map each of the elements $1, \ldots, 5$ to singletons in m', making the refinement the trivial abstraction $m' = \{\{1\}\mapsto k_1, \{2\}\mapsto k_2, \{3\}\mapsto k_3, \{4\}\mapsto k_4, \{5\}\mapsto k_5\}$.

As for the other spurious answer set $\hat{I}_3 = \{a(k), c(k), d(k), b(k,k)\}$ of Π^m (from Example 3.21), the minimal answer set of $\Pi_{debug} \cup Q^m_{\hat{I}_3}$ is $\{\mathit{ab\_deact}_{r4}(1)\}$ resulting in the hint atom $\mathit{refine}(1)$. When the abstraction mapping is refined to $m' = \{\{1\}\mapsto k_1, \{2,3,4,5\}\mapsto k_2\}$, the spurious answer set no longer appears.

Note that obtaining some *refine* atom during the correctness checking is not guaranteed whenever $\hat{I}$ is spurious, as the *ab* atoms may contain only domain elements that are mapped to singleton clusters. In this case, another abstract answer set $\hat{I}'$ of Π^m can be picked for the correctness checking.

5. Multi-dimensional domain abstraction

With the methods for abstraction and refinement in Sections 3 and 4 at hand, we are well equipped to run a CEGAR-style abstraction and refinement procedure for answer set programs. As we have seen from Proposition 3.8, it is possible to deal with sorts, which is important for some practical applications. We can construct an abstract program over multiple sorts in the manner of cartesian abstraction, which is achieved by doing abstraction over the sorts one at a time. However, this has the drawback that we cannot take certain interdependencies among the sorts into account, which in some scenarios may be needed. We illustrate this need on some examples and will then present an alteration of the abstraction method that can take interdependencies between sorts into account.

Example 5.1 *(Example 3.30 ctd).* An interesting abstraction would be to assign a color cluster $\widehat{rgb}$ only for the nodes $\{4,5,6\}$, which are clustered to a node $\hat{4}$, while for nodes $\{1,2,3\}$ the original colors are considered (see Fig. 9). Such an abstraction cannot be achieved with a cartesian style abstraction, since the color cluster $\widehat{rgb}$ is only meant to be considered for the node cluster $\hat{4}$. Thus, the desired abstraction can only be defined with a multi-dimensional mapping $m : D_n \times D_c \to \widehat{D}_n \times \widehat{D}_c$ as follows:

$$m(i,j) = \begin{cases} (i,j) & i \in \{1,2,3\},\ j \in \{\mathit{red}, \mathit{green}, \mathit{blue}\} \\ (\hat{4}, \widehat{rgb}) & i \in \{4,5,6\},\ j \in \{\mathit{red}, \mathit{green}, \mathit{blue}\} \end{cases}.$$

To further motivate the need for multi-dimensionality, we consider grid-cell domains, which are commonly used.

Example 5.2 *(grid-cell domains).* Usually the grid-cells are represented by using two sorts *row* and *column*. The following rules show the part of a Sudoku encoding that guesses an assignment of symbols to the cells and ensures that each cell has a number.

$$\begin{aligned} \{\mathit{sol}(X,Y,N)\} &\leftarrow \mathit{not\ occupied}(X,Y), \mathit{num}(N), \mathit{row}(X), \mathit{column}(Y). \\ \mathit{hasNum}(X,Y) &\leftarrow \mathit{sol}(X,Y,N), \mathit{row}(X), \mathit{column}(Y). \\ \bot &\leftarrow \mathit{not\ hasNum}(X,Y), \mathit{row}(X), \mathit{column}(Y). \end{aligned} \tag{57}$$

Further constraints ensure that cells in the same column (58) or same row (59) do not contain the same symbol.

$$\bot \leftarrow \mathit{sol}(X,Y_1,M), \mathit{sol}(X,Y_2,M), Y_1 < Y_2. \tag{58}$$

$$\bot \leftarrow \mathit{sol}(X_1,Y,M), \mathit{sol}(X_2,Y,M), X_1 < X_2. \tag{59}$$

A further more involved constraint (cf. Appendix B.2) ensures that the cells in the same sub-region also satisfy this.

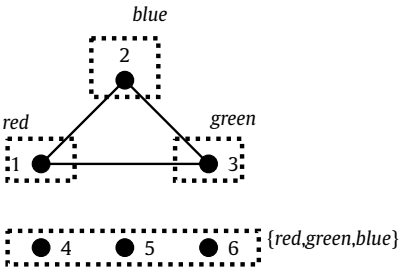


Fig. 9. Joint abstraction of nodes and colors.

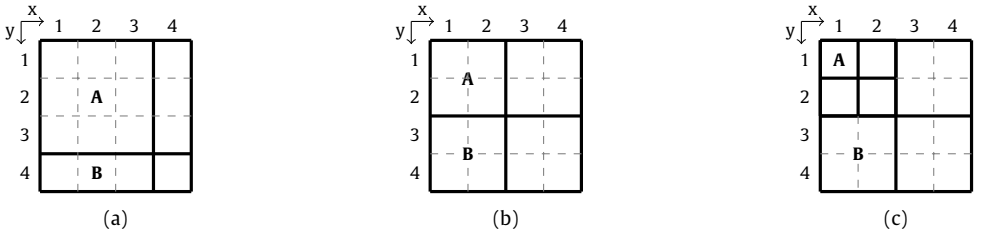


Fig. 10. Abstractions over grid-cells.

An abstraction over the grid-cells would be to cluster the rows and columns together in order to define an abstract grid-cell. Although abstraction over the sorts one at a time can achieve certain abstract cell structures, to obtain more sophisticated abstractions these sorts must be *jointly* abstracted. Consider for example the abstractions in Fig. 10. Those in Figs. 10a-10b can be achieved by independent mappings over the rows and columns such as $m_{row} = m_{col} = \{\{123\} \mapsto a_{123}, \{4\} \mapsto a_4\}$ and $m_{row} = m_{col} = \{\{12\} \mapsto a_{12}, \{34\} \mapsto a_{34}\}$. For a given program Π, one can construct the abstract program $(\Pi^{m_{row}})^{m_{col}}$. However to achieve Fig. 10c, rows and columns must be *jointly* abstracted. While the cells $(a_i, b_j), 1 \leq i, j \leq 2$ are singletons mapped from (i, j), the other abstract regions are only given by

$$m_{row,col}(x, y) = \begin{cases} (a_{12}, b_{34}) & x \in \{1, 2\}, y \in \{3, 4\} \\ (a_{34}, b_{12}) & x \in \{3, 4\}, y \in \{1, 2\} \\ (a_{34}, b_{34}) & x \in \{3, 4\}, y \in \{3, 4\} \end{cases} \tag{60}$$

Observe that the abstract row a_{12} describes a cluster that abstracts over the individual abstract rows a_1, a_2. The original rows $\{1, 2\}$ are mapped to $\{a_{12}\}$ only in combination with columns $\{3, 4\}$, otherwise they are mapped to $\{a_1, a_2\}$.

5.1. Existential abstraction on relations

The abstraction method described in Section 3.1.2 aims at keeping the built-ins in the abstract program and finds a way to handle their different behavior in the abstract domain. However, this approach cannot be used to achieve the above mentioned multi-dimensional abstraction. Consider the rule (58) standardized apart over rows and columns, thus having relations $X_1 = X_2$ and $Y_1 < Y_2$. If for the mapping $m_{row,col}$ in Fig. 10c these relations are lifted following Section 3.1.2, while the relation over the y-axis is still defined (as A is located above of B), i.e., $A_Y \leq B_Y$, the relation $A_X = B_X$ is unclear as the abstract clusters for X-values are different due to different levels of abstraction.

To tackle this issue, an alternative abstraction method is needed that also abstracts over the built-in relations and reasons over the abstracted relation (in the abstract domain). This leads us to a notion of abstraction that is similar in spirit to so called existential abstraction [25] and allows us to introduce domain mappings over multiple sorts such as

$$m : D_1 \times \ldots \times D_n \to \widehat{D}_1 \times \ldots \times \widehat{D}_n,$$

and to handle relations over different levels of abstraction.

For this, we introduce an abstract relation $\widehat{rel}$ for a k-ary relation rel as follows:

$$(\forall_{i=1}^{k} \hat{d}_i \in \widehat{D})\, \widehat{rel}(\hat{d}_1, \ldots, \hat{d}_k) \Leftrightarrow \exists_{i=1}^{k} x_i \in m^{-1}(\hat{d}_i).rel(x_1, \ldots, x_k). \tag{61}$$

$$(\forall_{i=1}^{k} \hat{d}_i \in \widehat{D})\, neg\_\widehat{rel}(\hat{d}_1, \ldots, \hat{d}_k) \Leftrightarrow \exists_{i=1}^{k} x_i \in m^{-1}(\hat{d}_i).\neg rel(x_1, \ldots, x_k). \tag{62}$$

I.e., $\widehat{rel}(\hat{d}_1, \ldots, \hat{d}_k)$ is true if for some corresponding original values the original relation holds; the opposite of $\widehat{rel}(\hat{d}_1, \ldots, \hat{d}_k)$, i.e. $neg\_\widehat{rel}(\hat{d}_1, \ldots, \hat{d}_k)$, is true otherwise. Notably, both versions may hold simultaneously, depending on the domain clusters.

Example 5.3 *(Example 3.1 ctd).* For the mapping $\{1,\ldots,5\}\mapsto k$ the abstract relation $k \widehat{\leq} k$ holds true, as $X \leq Y$ for all X, Y mapped to k. Both $k \widehat{=} k$ and $k\, neg\_\widehat{=} k$ hold true, as $X_1 = X_2$ holds only for some X_1, X_2 values mapped to k.

Notice that having both *rel* and *neg_rel* hold means an uncertainty on the truth value of the relation in the abstract clusters. This brings us to determining the types of the relations over the abstract clusters, similar as before.

Abstract relation types. The following cases $\tau_{\mathrm{I}} - \tau_{\mathrm{III}}$ occur in a mapping for the abstract relation predicates $\widehat{rel}(\hat{d}_1,\ldots,\hat{d}_k)$ and $\widehat{neg\_rel}(\hat{d}_1,\ldots,\hat{d}_k)$:

$$\begin{aligned}
&\tau_{\mathrm{I}}^{\widehat{rel}}(\hat{d}_1,\ldots,\hat{d}_k): && \widehat{rel}(\hat{d}_1,\ldots,\hat{d}_k) \wedge not\ \widehat{neg_rel}(\hat{d}_1,\ldots,\hat{d}_k)\\
&\tau_{\mathrm{II}}^{\widehat{rel}}(\hat{d}_1,\ldots,\hat{d}_k): && \widehat{neg_rel}(\hat{d}_1,\ldots,\hat{d}_k) \wedge not\ \widehat{rel}(\hat{d}_1,\ldots,\hat{d}_k)\\
&\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1,\ldots,\hat{d}_k): && \widehat{rel}(\hat{d}_1,\ldots,\hat{d}_k) \wedge \widehat{neg_rel}(\hat{d}_1,\ldots,\hat{d}_k)
\end{aligned} \tag{63}$$

Type I is the case where the abstraction does not cause uncertainty for the relation, thus the rules that contain $\widehat{rel}$ with type I can remain the same in the abstract program. Type II shows the cases where $\widehat{rel}$ does not hold in the abstract domain. Type III is the case of uncertainty, which needs to be dealt with when creating the abstract rules. To ensure that an over-approximation is achieved, the head of the respective rule will be changed into a choice.

For an abstraction m, we compute the set $\mathcal{T}_{m_\exists}$ of all atoms $\tau_\iota^{\widehat{rel}}(\hat{d}_1,\ldots,\hat{d}_k)$ where $\iota \in \{\mathrm{I},\mathrm{II},\mathrm{III}\}$ is the type of $\widehat{rel}(\hat{d}_1,\ldots,\hat{d}_k)$ for m.

5.1.1. Abstraction procedure

For simplicity and ease of presentation, we consider programs with rules having (i) a single relation atom; and (ii) no cyclic dependencies between non-ground literals.

Definition 5.1 *(rule abstraction).* Given a rule $r:\ \alpha \leftarrow B^{std}(r), rel(t_1,\ldots,t_k)$ and a domain mapping m, the set $r_\exists^m$ contains the following rules.

(a) $m(\alpha) \leftarrow m(B^{std}(r)), \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{t}_1,\ldots,\hat{t}_k).$
(b) $\{m(\alpha)\} \leftarrow m(B^{std}(r)), \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{t}_1,\ldots,\hat{t}_k).$
(c) For all $L \subseteq B^{std,-}(r)$:

$$\begin{aligned}
&\{m(\alpha)\} \leftarrow m(B_L^{sh}(r)), \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{t}_1,\ldots,\hat{t}_k), not\ isSingleton(m(t)).\\
&\{m(\alpha)\} \leftarrow m(B_L^{sh}(r)), \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{t}_1,\ldots,\hat{t}_k), not\ isSingleton(m(t)).
\end{aligned} \qquad t \in arg(\alpha_i), \alpha_i \in L$$

where $B_L^{sh}(r) = B^{std,+}(r) \cup L, not\ B^{std,-}(r) \setminus L$.

The idea is to introduce guesses when there is an uncertainty over the relation holding in the abstract domain (b), or over the negated atoms due to the abstract clusters (c) (by considering all combinations of the negative literals), and otherwise just abstracting the rule (a).

The abstraction procedure introduced in Definition 5.1 obtains semantically the same abstract program as in Definition 3.4 for rules of form

$$\alpha \leftarrow B^{std}(r), rel(t_1,t_2).$$

where $rel(t_1,t_2) = t_1 \circ t_2$ is a binary relation with $\circ \in \{=,\neq,<,\leq,>,\geq\}$.

Definition 5.2 *(existential abstract program $\Pi_\exists^m$).* Given a program Π and a domain mapping m, we denote by

$$\Pi_\exists^m = \bigcup_{r:\ \alpha \leftarrow B^{std}(r), rel(t_1,t_2) \in \Pi} r_\exists^m \cup \{x.\mid x \in \mathcal{T}_{m_\exists}\} \cup \{m(p(\vec{c})).\mid p(\vec{c}). \in \Pi\} \cup \{isSingleton(\hat{d}) \mid |m^{-1}(\hat{d})| = 1\}. \tag{64}$$

the program obtained from Π under existential abstraction using m.

We then have

Theorem 5.1. *For any domain mapping m of a (standardized apart) program Π with rules having a single relation atom and no cyclic dependencies between non-ground literals, $AS(\Pi^m)$ and $AS(\Pi_\exists^m)$ coincide (modulo auxiliary atoms).*

A generalization to multiple relation atoms and handling cyclic dependencies by removing the restrictions (i)-(ii) can be done similarly as in cases (G-II) and (G-III) of Section 3.2.

Example 5.4 *(Example 3.1 ctd).* For the program Π in (4)-(6) with the choice rules (28)-(29) and $m = \{\{1,\ldots,5\}\mapsto k\}$, the program $\Pi^m_\exists$ is:

$$c(X) \leftarrow not\ d(X), dom(X).$$
$$\{c(X)\} \leftarrow not\ isSingleton(X), dom(X). \tag{65}$$
$$d(X) \leftarrow not\ c(X), dom(X).$$
$$\{d(X)\} \leftarrow not\ isSingleton(X), dom(X). \tag{66}$$
$$b(X,Y) \leftarrow a(X), d(Y).$$
$$e(X) \leftarrow c(X), a(Y), \tau_{\mathrm{I}}^{\hat{\neq}}(X,Y).$$
$$\{e(X)\} \leftarrow c(X), a(Y), \tau_{\mathrm{III}}^{\hat{\neq}}(X,Y).$$
$$\bot \leftarrow b(X,Y), e(X_1), \tau_{\mathrm{I}}^{\hat{=}}(X,X_1).$$

with the abstract fact $a(k)$; furthermore, we have $\mathcal{T}_{m_\exists} = \{\tau_{\mathrm{III}}^{\hat{\neq}}(k,k), \tau_{\mathrm{III}}^{\hat{=}}(k,k)\}$. Note that the atoms $d(X)$ and $c(X)$ are omitted in (65) and (66) respectively, as they are involved in a negative cycle, similar as in rules (35) and (37) of Example 3.19.

The abstract program is similar to the one constructed by lifting the relations in Example 3.19. As can be easily checked, the programs have modulo the auxiliary atoms the same abstract answer sets.

We note that the previous refinement methods can be applied to the abstract program constructed in this way as well, since nothing changes with regard to how this program can be refined. Furthermore, we observe that for treating n-ary relations where $n > 2$, we can modify Definition 5.1 to create finer abstractions.

Example 5.5. Consider the argument Z of the following rule involving addition:

$$r : e(Z) \leftarrow c(X), a(Y), Z = X + Y. \tag{67}$$

We denote $Z = X + Y$ with the relation $plus(X,Y,Z)$. Regarding the arguments, we have $arg(e(Z)) \cap arg(plus(X,Y,Z)) = \{Z\} \neq \emptyset$ while $arg(e(Z)) \cap \{X,Y\} = \emptyset$, where X, Y are the shared arguments of the body literals with the relation $plus$, i.e., $arg(B(r)) \cap arg(plus(X,Y,Z)) = \{X,Y\}$. Consider the mapping $m : \{1\} \mapsto a_1, \{2,3\} \mapsto a_{23}$, $\{4,5\} \mapsto a_{45}$ and $X{=}a_1, Y{=}a_1$. For the abstract relation $\widehat{plus}$, both $\widehat{plus}(a_1,a_1,a_{23})$ and $\widehat{neg\_plus}(a_1,a_1,a_{23})$ hold true, due to $1+1=2$ and $1+1\neq 3$. As Z is not used in the body literals, it does not cause uncertainties for applying the rule in the abstraction, which is caught by

$$e(Z) \leftarrow c(X), a(Y), \tau_{\mathrm{III}}^{\widehat{rel}(X,Y,Z)}, isSingleton(X), isSingleton(Y).$$

In general, by adding in Definition 5.1 the rule

$$m(\alpha) \leftarrow m(B^{std}(r)), \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{t}_1,\ldots,\hat{t}_k), \textstyle\bigwedge_{\hat{t}_i \in arg_i(rel)\setminus arg(\alpha)} isSingleton(\hat{t}_i).$$
$$\text{if } arg(\alpha) \cap arg(rel) \neq \emptyset \text{ and } arg(\alpha) \cap arg(B^{std}(r) \cap arg(rel)) = \emptyset.$$

the guess in (b) can be avoided, if all arguments of rel not involved in the head l are singleton clusters.

The use of abstract relations opens a wide range of possible applications, as it simplifies the use of a given program without preprocessing it to match the restrictions over the forms of the relations for the previous abstraction method.

5.1.2. Computing joint abstract relation types

Abstract relations can be easily employed with abstraction mappings over several sorts in the domain as $m : D_1 \times \cdots \times D_n \to \widehat{D}_1 \times \cdots \times \widehat{D}_n$. If a rule has relations over the sorts, a joint abstract relation combining them must be computed. We show an example of grid-cell abstraction for illustration.

Example 5.6 *(abstracting grid-cells).* Consider the relations $rel_1(X_1,X_2)$: $X_1 = X_2$ and $rel_2(Y_1,Y_2)$: $Y_1 < Y_2$ for $X_1, X_2 \in row$, Y_1, $Y_2 \in column$, from standardizing apart the variables in (58). The rules to compute the types $\tau_{\mathrm{I}}^{\widehat{rel}}$, $\tau_{\mathrm{III}}^{\widehat{rel}}$, where $\widehat{rel}$ combines rel_1 and rel_2, are as follows:

1. Define the abstract relations. This step corresponds to the existential abstraction (61).

$$\widehat{rel_1}((\hat{X}_1,\hat{Y}_1),(\hat{X}_2,\hat{Y}_2)) \leftarrow rel_1(X_1,X_2), m((X_1,Y_1),(\hat{X}_1,\hat{Y}_1)), m((X_2,Y_2),(\hat{X}_2,\hat{Y}_2)).$$
$$\widehat{rel_2}((\hat{X}_1,\hat{Y}_1),(\hat{X}_2,\hat{Y}_2)) \leftarrow rel_2(Y_1,Y_2), m((X_1,Y_1),(\hat{X}_1,\hat{Y}_1)), m((X_2,Y_2),(\hat{X}_2,\hat{Y}_2)).$$

$$\widehat{rel}_i(\hat{c}_1,\ldots,\hat{c}_k) \leftarrow rel_i(d_1^i,\ldots,d_k^i), m((d_1^1,\ldots,d_1^{n_2}),\hat{c}_1),\ldots,m((d_k^1,\ldots,d_k^{n_2}),\hat{c}_k). \qquad i=1,\ldots n_1$$

$$\widehat{neg\_rel}_i(\hat{c}_1,\ldots,\hat{c}_k) \leftarrow \neg rel_i(d_1^i,\ldots,d_k^i), m((d_1^1,\ldots,d_1^{n_2}),\hat{c}_1),\ldots,m((d_k^1,\ldots,d_k^{n_2}),\hat{c}_k). \qquad i=1,\ldots n_1$$

$$\tau_{\mathrm{I}}^{\widehat{rel}_i}(\hat{c}_1,\ldots,\hat{c}_k) \leftarrow \widehat{rel}_i(\hat{c}_1,\ldots,\hat{c}_k), not\ \widehat{neg\_rel}_i(\hat{c}_1,\ldots,\hat{c}_k). \qquad i=1,\ldots,n_1$$

$$\tau_{\mathrm{II}}^{\widehat{rel}_i}(\hat{c}_1,\ldots,\hat{c}_k) \leftarrow not\ \widehat{rel}_i(\hat{c}_1,\ldots,\hat{c}_k), \widehat{neg\_rel}_i(\hat{c}_1,\ldots,\hat{c}_k). \qquad i=1,\ldots,n_1$$

$$\tau_{\mathrm{III}}^{\widehat{rel}_i}(\hat{c}_1,\ldots,\hat{c}_k) \leftarrow \widehat{rel}_i(\hat{c}_1,\ldots,\hat{c}_k), \widehat{neg\_rel}_i(\hat{c}_1,\ldots,\hat{c}_k). \qquad i=1,\ldots,n_1$$

$$\tau_{\mathrm{I}}^{\widehat{rel_1,\ldots,rel_{n_1}}}(\hat{c}_1,\ldots,\hat{c}_k) \leftarrow \tau_{\mathrm{I}}^{\widehat{rel}_1}(\hat{c}_1,\ldots,\hat{c}_k),\ldots,\tau_{\mathrm{I}}^{\widehat{rel}_{n_1}}(\hat{c}_1,\ldots,\hat{c}_k).$$

$$\tau_{\mathrm{III}}^{\widehat{rel_1,\ldots,rel_{n_1}}}(\hat{c}_1,\ldots,\hat{c}_k) \leftarrow \tau_{\mathrm{III}}^{\widehat{rel}_i}(\hat{c}_1,\ldots,\hat{c}_k), \textstyle\bigwedge_{j=1:j\neq i}^{n_2} not\ \tau_{\mathrm{II}}^{\widehat{rel}_j}(\hat{c}_1,\ldots,\hat{c}_k). \qquad i=1,\ldots,n_1$$

Fig. 11. Computation of multi-dimensional relation types ($\hat{c}_j=(\hat{d}_j^1,\ldots,\hat{d}_j^{n_2}), 1\le j\le k$).

The negations $\widehat{neg\_rel_1}, \widehat{neg\_rel_2}$ are computed similarly as (62).

$$\widehat{neg\_rel_1}((\hat{X}_1,\hat{Y}_1),(\hat{X}_2,\hat{Y}_2)) \leftarrow \neg rel_1(X_1,X_2), m((X_1,Y_1),(\hat{X}_1,\hat{Y}_1)), m((X_2,Y_2),(\hat{X}_2,\hat{Y}_2)).$$
$$\widehat{neg\_rel_2}((\hat{X}_1,\hat{Y}_1),(\hat{X}_2,\hat{Y}_2)) \leftarrow \neg rel_2(Y_1,Y_2), m((X_1,Y_1),(\hat{X}_1,\hat{Y}_1)), m((X_2,Y_2),(\hat{X}_2,\hat{Y}_2)).$$

2. Compute the types of each abstract relation $\widehat{rel}_i, i\in\{1,2\}$ with the objects $\hat{C}_i=(\hat{X}_i,\hat{Y}_i), i\in\{1,2\}$ as (63).

$$\tau_{\mathrm{I}}^{\widehat{rel}_i}(\hat{C}_1,\hat{C}_2) \leftarrow \widehat{rel}_i(\hat{C}_1,\hat{C}_2), not\ \widehat{neg\_rel}_i(\hat{C}_1,\hat{C}_2).$$
$$\tau_{\mathrm{II}}^{\widehat{rel}_i}(\hat{C}_1,\hat{C}_2) \leftarrow not\ \widehat{rel}_i(\hat{C}_1,\hat{C}_2), \widehat{neg\_rel}_i(\hat{C}_1,\hat{C}_2).$$
$$\tau_{\mathrm{III}}^{\widehat{rel}_i}(\hat{C}_1,\hat{C}_2) \leftarrow \widehat{rel}_i(\hat{C}_1,\hat{C}_2), \widehat{neg\_rel}_i(\hat{C}_1,\hat{C}_2).$$

3. Compute the types of the joint abstract relation $\widehat{rel}$ over $\widehat{rel}_i, i\in\{1,2\}$:

$$\tau_{\mathrm{I}}^{\widehat{rel}}(\hat{C}_1,\hat{C}_2) \leftarrow \tau_{\mathrm{I}}^{\widehat{rel1}}(\hat{C}_1,\hat{C}_2), \tau_{\mathrm{I}}^{\widehat{rel2}}(\hat{C}_1,\hat{C}_2).$$
$$\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{C}_1,\hat{C}_2) \leftarrow not\ \tau_{\mathrm{II}}^{\widehat{rel1}}(\hat{C}_1,\hat{C}_2), \tau_{\mathrm{III}}^{\widehat{rel2}}(\hat{C}_1,\hat{C}_2).$$
$$\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{C}_1,\hat{C}_2) \leftarrow \tau_{\mathrm{III}}^{\widehat{rel1}}(\hat{C}_1,\hat{C}_2), not\ \tau_{\mathrm{II}}^{\widehat{rel2}}(\hat{C}_1,\hat{C}_2).$$

The mapping (60) shown in Fig. 10c gives the types $\tau_{\mathrm{I}}^{\widehat{rel}}((a_1,b_1),(a_1,b_2))$, $\tau_{\mathrm{I}}^{\widehat{rel}}((a_2,b_1),(a_2,b_2))$ and $\tau_{\mathrm{III}}^{\widehat{rel}}$ for the remaining abstract pairs.

Fig. 11 presents the multi-dimensional case, that is computing abstract k-tuple relations for given relations $rel_1,\ldots,rel_{n_1}$ over variables from $\widehat{D}_1,\ldots,\widehat{D}_{n_2}$. We assume for simplicity a uniform arity k.

Note that for the joint abstract relation $\widehat{rel}$, type $\tau_{\mathrm{II}}^{\widehat{rel}}$ computation is not needed, as the abstract rule construction only deals with types I and III. To emphasize the abstracted relations, we may denote $\widehat{rel}$ in $\tau^{\widehat{rel}}$ with the combination of the relations the abstract relation is built on; e.g., for the joint relation type of $rel_1(X_1,X_2)\colon X_1=X_2$ and $rel_2(Y_1,Y_2)\colon Y_1<Y_2$ we write $\tau_{\mathrm{I}}^{=,<}, \tau_{\mathrm{III}}^{=,<}$.

The multi-dimensional abstraction constructs an abstract structure, i.e., *object*, over the abstracted sorts where not all combinations of the abstract sorts yield a valid object. To illustrate this, in Example 5.1 the color cluster $\widehat{rgb}$ can only be considered with the node cluster $\hat{4}$. This also needs to be taken into account when constructing the abstract program.

Example 5.7. The abstract program for Sudoku (57)-(59), where the occurrences of $row(X)$, $column(Y)$ are replaced by $cell(X,Y)$, is as follows.

$$hasNum(X,Y) \leftarrow sol(X,Y,N), cell(X,Y).$$
$$\{sol(X,Y,N)\} \leftarrow not\ occupied(X,Y), num(N), cell(X,Y).$$
$$\{sol(X,Y,N)\} \leftarrow occupied(X,Y), num(N), not\ isSingleton(X), cell(X,Y).$$
$$\{sol(X,Y,N)\} \leftarrow occupied(X,Y), num(N), not\ isSingleton(Y), cell(X,Y).$$
$$\bot \leftarrow not\ hasNum(X,Y), cell(X,Y).$$
$$\bot \leftarrow sol(X_1,Y_1,M), sol(X_2,Y_2,M), \tau_{\mathrm{I}}^{=,<}(X_1,Y_1,X_2,Y_2), cell(X,Y_1), cell(X_2,Y_2).$$
$$\bot \leftarrow sol(X_1,Y_1,M), sol(X_2,Y_2,M), \tau_{\mathrm{I}}^{<,=}(X_1,Y_1,X_2,Y_2), cell(X_1,Y), cell(X_2,Y_2).$$

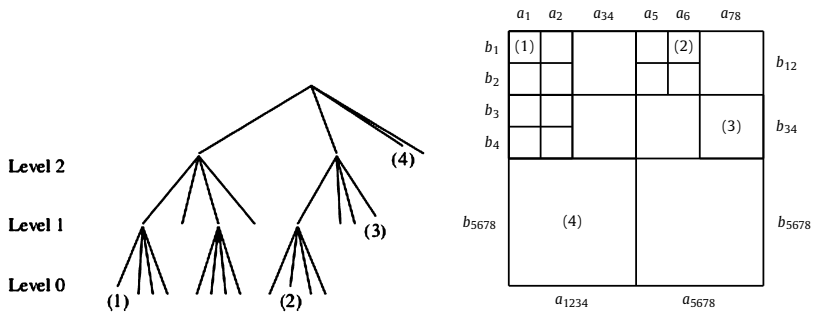


Fig. 12. Quad-tree representation for regions.

5.1.3. Refinement of multi-dimensional abstractions

The refinement method introduced in Section 4 can be used for multi-dimensional abstractions as well. For this to work, we need the k-tuples, on which the abstraction will be made, to occur in the rules. For example, for a 2-dimensional abstraction, the pairs X, Y should appear in the rules together. This will ensure that when the meta-programs are constructed as in Definitions 4.4 and 4.5, the abnormality atoms will contain these tuples.

The non-ground query $Q^m_{\hat{I}}$ in Definition 4.3 must be updated with mappings of form $m((X_1, Y_1), (\hat{X}_1, \hat{Y}_1))$ instead of $m(X_1, \hat{X}_1)$. The program Π_{hint} that gets hints $refine(c_1, \ldots, c_n)$ about which cluster to refine in Definition 4.7 also needs to be updated, so that it can be used to refine the multi-dimensional abstraction mapping. For example, in case of a 2-dimensional abstraction mapping, for $c_{x_i}, c_{y_i} \in arg(ab\_deact_{n_r}(c_{x,y}))$, where $c_{x,y} = c_{x_1}, c_{y_1}, \ldots, c_{x_n}, c_{y_n}$, we will have

$$refine(c_{x,y}) \leftarrow ab\_deact_{n_r}(c_{x,y}), m(c_{x_i}, c_{y_i}, a_{x_i}, a_{y_i}), not\ isSingleton(a_{x_i}). \tag{68}$$

$$refine(c_{x,y}) \leftarrow ab\_deact_{n_r}(c_{x,y}), m(c_{x_i}, c_{y_i}, a_{x_i}, a_{y_i}), not\ isSingleton(a_{y_i}). \tag{69}$$

In fact, if we focus only on the abstract region that needs to be refined, we may use atoms $refine(a_{x_i}, a_{y_i})$ in the heads of the rules (68) and (69), and then decide on a refinement for the region (a_{x_i}, a_{y_i}). A similar change will be needed for $ab\_deactCons_{n_r}$ and ab_act.

After these changes the refinement method can be applied to multi-dimensional abstractions. More information on our approach to deciding on a refinement in the implementation is given in Section 6.2.2.

5.2. Quad-tree abstraction

Grid-cell environments are a particular type of environment which describes a structure. For problems over grid-cells, it is often the case that certain parts of the environment are crucial to finding a solution. In order to obtain an abstraction over a grid-cell that allows to adjust its granularity, multi-dimensionality must be considered. Multi-dimensional abstraction allows us to express abstractions where one sort in the domain (e.g., an X coordinate) is abstracted depending on its context, i.e., depending on a second sort in the domain it occurs with (e.g., a Y coordinate).

For a systematic refinement of abstractions on grid-cell environments, we consider a generic quad-tree representation (Fig. 12), which is a concept used, e.g., in path planning [69]. Initially, an environment may be abstracted to four regions of $n/2 \times n/2$ grid-cells each. This amounts to a tree with four leaf nodes that correspond to the main regions. Each region then contains 4 leaves of smaller regions. The leaves of the quad-tree are then the original cells of the grid-cell at level 0. A refinement of a region amounts to dividing it into four subregions, i.e., sprouting the respective node to four children. Given the original X and Y coordinates $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, respectively, we represent the coordinates of an abstract region with level $log_2(k{+}1)$, for $0 \leq k < n$, defined over the cells within the coordinates $a_i, \ldots, a_{i+k}$ and $b_j, \ldots, b_{j+k}$, respectively, by the shorthand notation $(a_{i\ldots i+k}, b_{j\ldots j+k})$.

Starting with an initial abstraction of level $log_2(n)$, using quad-tree split operations as abstraction refinement operations, we can automatically search for suitable quad-tree-structured abstractions in grids (see Section 6.2.2). Importantly, multi-dimensional abstraction refinement is *structure aware*: refining one of the squares of a quad-tree (e.g., area (3) in Fig. 12) maintains the structure of the abstraction of all other squares.

We illustrate next how such a structure can be used to adjust the granularity of the abstraction over the grid-cell.

Example 5.8 *(Reachability).* Suppose one wants to check whether all cells are reachable from a given starting point in a grid with obstacles. In case there are unreachable cells, this is due to obstacles separating them from other cells. For a person, a glance over the area with the obstacles will be sufficient to realize that some cells are unreachable. The rules below compute the obstacle-free cells (i.e., *points*) that are reachable from the starting point; an additional constraint (75) checks whether all points are reachable.

$$point(X, Y) \leftarrow not\ obsAt(X, Y), row(X), column(Y). \tag{70}$$

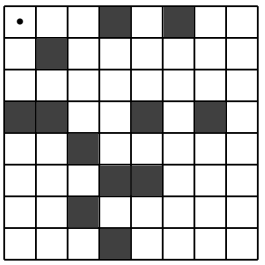
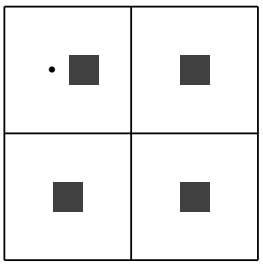


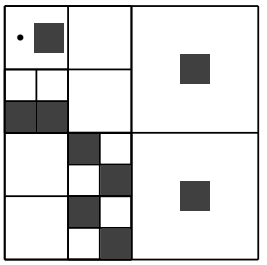
Fig. 13. Original grid-cell domain.

(a) Initial abstraction



$start(a_{1234}, b_{1234}).$
$obsAt(a_{1234}, b_{1234}).$
$obsAt(a_{1234}, b_{5678}).$
$obsAt(a_{5678}, b_{1234}).$
$obsAt(a_{5678}, b_{5678}).$

(b) Distinguishing the obstacles that cause unreachability



$start(a_{12}, b_{12}).$
$obsAt(a_1, b_4).\ obsAt(a_2, b_4).$
$obsAt(a_3, b_5).\ obsAt(a_4, b_6).$
$obsAt(a_3, b_7).\ obsAt(a_4, b_8).$
$obsAt(a_{12}, b_{12}).$
$obsAt(a_{5678}, a_{1234}).$
$obsAt(a_{5678}, a_{5678}).$

Fig. 14. Abstractions over the grid-cell domain with obstacles in Fig. 13.

$$reachable(X, Y) \leftarrow start(X, Y). \tag{71}$$
$$reachable(X_1, Y_1) \leftarrow reachable(X, Y), point(X_1, Y_1), neighbor(X, Y, X_1, Y_1). \tag{72}$$
$$neighbor(X, Y, X_1, Y) \leftarrow |X - X_1| = 1, column(Y). \tag{73}$$
$$neighbor(X, Y, X, Y_1) \leftarrow |Y - Y_1| = 1, row(X). \tag{74}$$
$$\bot \leftarrow point(X, Y), not\ reachable(X, Y), row(X), column(Y). \tag{75}$$

Fig. 13 shows an instance of a grid-cell domain with obstacles; the program is unsatisfiable on it due to the unreachable cells in the lower left area. Fig. 14 shows two abstractions over the grid-cells, including the abstracted start and obstacle positions, the initial abstraction dividing the grid-cell into 4 regions (Fig. 14a), and an abstraction that distinguishes the area which shows the obstacles causing unreachable cells in the lower-left corner (Fig. 14b). Appendix B.3 provides a detailed example of applying automatic refinement (Section 5.1.3) and the mDASPAR tool (Section 6.2.2) to this example.

6. Implementation and evaluation

6.1. Overall methodology

The abstraction and refinement method for domain abstraction is shown in Algorithm 2. In the initialization phase, the abstract program is constructed for the mapping m depending on the *mode* of abstraction (lifted or existential) on the built-ins (Line 2); the relation types are also computed in this step. After constructing the debugging program (Line 3), some abstract answer set is computed (Lines 5), and its concreteness checked with a call to the function *getASWithMinAbAtoms* on the non-ground query $Q^m_{\hat{I}}$ (Definition 4.3) for m and $\hat{I}$ (Line 6), which returns an answer set of $\Pi_{debug} \cup Q^m_{\hat{I}}$ with minimal number of *ab* atoms. If the latter is positive (i.e., $\hat{I}$ is spurious), m is refined given the *ab* atoms found in the check (Line 9) and the abstract program and the relation types are recomputed before looping back to evaluation. Among possible variations of the algorithm, we want to mention the following.

Abstraction over subdomains/sorts For abstraction over a particular subdomain resp. sort, Algorithm 2 can be extended with a predicate name s for the sort in the input and the computation of the abstract program, the relation types and the debugging program focused on the domain elements related with s.

Correctness checking for relevant atoms The correctness checking of abstract answer sets can be confined to the *relevant* atoms describing the solution by constructing $Q^m_{\hat{I}}$ (Line 6) only for these atoms. An abstract answer set will then pass as concrete as long as it describes a concrete solution with respect to them.

Algorithm 2: *Domain-Abs&Ref*.

Input: program Π, domain mapping m_{init}, *mode* (lifted or existential)
Output: Π^m, a mapping m refining m_{init}, and an abstract answer set $\hat{I}$ of Π^m resp. $\emptyset$ if Π^m is unsatisfiable.

1 $m = m_{init}$;
2 $[\Pi^m, \mathcal{T}_m] = constructAbsProg(\Pi, m, mode)$;
3 $\Pi_{debug} = constructDebugProg(\Pi)$;
4 **while** $AS(\Pi^m \cup \mathcal{T}_m) \neq \emptyset$ **do**
5 Get $\hat{I} \in AS(\Pi^m \cup \mathcal{T}_m)$;
6 $C = getASWithMinAbAtoms(\Pi_{debug} \cup Q^m_{\hat{I}})$;
7 **if** $C|_{ab} = \emptyset$ **then** /* $\hat{I}$ is concrete */
8 **return** $\Pi^m, m, \hat{I}$
9 $m = decideRefinement(m, C|_{ab})$;
10 $[\Pi^m, \mathcal{T}_m] = constructAbsProg(\Pi, m, mode)$;
11 **return** $\Pi^m, m, \emptyset$

Diverse abstract answer sets The refinement decision may be made by looking at multiple abstract answer sets rather than a single one. Lines 5-8 are changed to collect the checking results $C_1, \ldots, C_n$ for abstract answer sets $I_1, \ldots, I_n$; then *decideRefinement* can use the collected results $C_1|_{ab}, \ldots, C_n|_{ab}$ (all assumed to be nonempty) for deciding on a refinement. We call **(v1)** the refinement approach where the answer set C_i with the least cost (aggregated from the collection) is picked, while refinement approach **(v2)** decides by choosing the *refine* atom that occurs most frequently in the answer sets $C_1, \ldots, C_n$.

6.2. Implementation

The methodology in Section 6.1 has been implemented in tools called DASPAR and mDASPAR based on clingo 5.2.2, Python and the Ouroboros debugging tool [93], whose MetaTranslator is exploited to obtain a reified program for which then the debugging program is constructed (cf. Section 4); negative cycles are merged as described in Section 3.2.3. More details are provided in the next subsections. The implementations are online available at http://www.kr.tuwien.ac.at/research/systems/abstraction/.

6.2.1. DASPAR

The program DASPAR supports abstraction from sorts, but the input program Π must adhere to certain restrictions: each variable in a rule must be guarded by a domain predicate; if the abstraction should be on some subset S of sorts, then the variables referring to the sorts in S must be standardized apart. For example, a rule of form

$$a(X) \leftarrow b(X, X_1), c(X_2), d(X_2), X \leq X_1.$$

needs to be converted into

$$a(X) \leftarrow b(X, X_1), c(X_2), d(X_3), X \leq X_1, X_2 = X_3, dom(X), dom(X_1), dom(X_2), dom(X_3).$$

with domain predicate *dom*. In order to support the case of having more than one relation, a syntactic change on the rule has to be made. These relations need to be combined into an auxiliary relation atom which represents the combination of the relations. The above rule needs to be converted into

$$a(X) \leftarrow b(X, X_1), c(X_2), d(X_3), leqEqu4(X, X_1, X_2, X_3).$$

where $leqEqu4(X, X_1, X_2, X_3)$ is an auxiliary atom which holds true whenever the respective relation holds true for its arguments. A basic set of auxiliary relation combinations are built into the tool, and more can easily be added.

DASPAR is invoked as follows.

```
python daspar.py prog mapping pred ref_type <focus_atoms>
```

Here `prog` contains the original program in the input format and the `mapping` the abstraction mapping information. DASPAR supports abstraction on one sort (see Section 6.2.2 for multi-dimensional tool mDASPAR), thus `pred` should be the name of the sort to the abstracted. The parameter `ref_type` allows to specify whether the refinement should respect an order relation (`1`) or not (`0`) (see Section 3.3.1). If `1` is given, the refinement step considers only splitting the domain, while when `0` is given the refinement step is unrestricted. The parameter `<focus_atoms>` is an optional input for projection in the correctness check (see Section 6.1).

DASPAR has different settings for picking abstract answer sets and for deciding on a refinement. For the former, by default the first computed answer set is picked. This can be changed to considering a diverse set of abstract answer sets. For deciding on a refinement, the two forms mentioned in Section 4 are implemented. Later, we evaluate the effects of having these different settings in the methodology on the achieved resulting abstractions.

```
                point(X,Y) :- not obsAt(X,Y), row(X), column(Y).
             reachable(X,Y) :- start(X,Y), row(X), column(Y).
          reachable(X1,Y1) :- reachable(X,Y), point(X1,Y1), neighbor(X2,Y2,X3,Y3),
                              equEqu4(X,X2,X1,X3), equEqu4(Y,Y2,Y1,Y3),
                              row(X), column(Y), row(X1), column(Y1),
                              row(X2), column(Y2), row(X3), column(Y3).
                           :- point(X,Y), not reachable(X1,Y1), X=X1, Y=Y1,
                              row(X), column(Y), row(X1), column(Y1).
neighbor(X,Y,X1,Y1) :- dist1(X,X1), Y=Y1, row(X), column(Y), row(X1), column(Y1).
neighbor(X,Y,X1,Y1) :- X=X1, dist1(Y,Y1), row(X), column(Y), row(X1), column(Y1).
```

Fig. 15. Input program with the rules (70)-(75). Subdomain predicates are `row` and `column`.

For practical purposes, sorts can be overlapping, provided that all occurrences of a sort are guarded by subdomain predicates. E.g., the blocksworld has sorts *block* and *time* which both can use integers. Note that this restriction is to aid the machine knowing about the relations of the arguments, which the user implicitly knows when encoding the problem. With this guidance, it becomes clear which arguments in the rule the abstraction should focus on.

6.2.2. mDASPAR

The program mDASPAR extends DASPAR to multi-dimensional domain abstraction. It handles 2-dimensional abstractions with a quad-tree style refinement process, and it can be applied to problems over cells in grids of size $n = 2^k$ for $k \geq 2$. We discuss some challenges of multi-dimensional abstractions that are tackled in the system.

Abstract objects A multi-dimensional abstraction creates abstract objects for tuples of concrete objects; not all combinations of the abstracted sorts, e.g., *row* and *column*, correspond to a valid object. To avoid such combinations, the constructed abstract program should comply to only using the abstract objects in the rules. For this, mDASPAR post-processes the abstract program and replaces the occurrence of the abstracted sorts with a new object name.

Note that for "grouping" objects automatically and correctly, the system needs some guidance. For a given encoding, humans are capable of detecting the cells implicitly, whereas a machine cannot do this readily. The user must provide it with some guidelines to recognize the objects, by adjusting the encoding so that the grids are explicitly shown. For this, we impose some syntactic restrictions on the input program, on which the post-processing technique relies.

Given two sorts s_1, s_2 for a 2-dimensional abstraction, the input program should adhere to the following restrictions in order to achieve a correct object naming:

(1) The rules should have atoms that contain pairs X, Y of variables where $X \in s_1, Y \in s_2$, and
(2) the subdomain predicates for sorts s_1, s_2 should be written in the order of the pairs.

If these restrictions are satisfied, then mDASPAR can correctly convert the sort names to the abstract object name *cell*. For example, $row(X_1), column(Y_1)$ is changed to $cell(X_1, Y_1)$.

Example 6.1 *(ctd).* The rule (72) will be standardized apart into

$$reachable(X_1, Y_1) \leftarrow \quad reachable(X, Y), point(X_1, Y_1), neighbor(X_2, Y_2, X_3, Y_3), \\ X = X_2, X_1 = X_3, Y = Y_2, Y_1 = Y_3.$$

Then the multiple relations related with a sort should be converted into an auxiliary relation atom:

$$reachable(X_1, Y_1) \leftarrow \quad reachable(X, Y), point(X_1, Y_1), neighbor(X_2, Y_2, X_3, Y_3), \\ equEqu4(X, X_2, X_1, X_3), equEqu4(Y, Y_2, Y_1, Y_3).$$

The subdomain predicates for the rule above also need to be written in a format where the pairs (X, Y), (X_1, Y_1), (X_2, Y_2), and (X_3, Y_3) appear together. Fig. 15 shows the resulting rules in the input program, including all subdomain predicates.

Relation type computation When abstracting a rule, mDASPAR gathers the relations in it related with the abstracted sorts and creates an abstract relation atom following the description in Section 5. The relation type facts (τ) are computed using auxiliary programs.

The program mDASPAR is invoked similarly to DASPAR, but with an additional parameter `size` which is the number n. The next example shows the input format of mDASPAR and the created abstract program.

Example 6.2 *(ctd).* Fig. 15 shows the input program for mDASPAR with the rules (70)-(75) where variables are standardized apart. The program constructed for abstracting over the sorts $row, column$ is shown in Fig. 16, where the occurrence of the sorts are renamed with a new object *cell*. The rules of the original program are numbered $(r_1, \ldots, r_6)$ and the relation

```
          point(X,Y) :- cell(X,Y), not obsAt(X,Y).
      { point(X,Y) } :- cell(X,Y), obsAt(X,Y),  not isSingleton(X).
      { point(X,Y) } :- cell(X,Y), obsAt(X,Y),  not isSingleton(Y).
     reachable(X,Y) :- start(X,Y), cell(X,Y).
   reachable(X1,Y1) :- reachable(X,Y), point(X1,Y1), neighbor(X2,Y2,X3,Y3),
                       cell(X2,Y2), cell(X3,Y3), cell(X,Y), cell(X1,Y1),
                       relr3(X,Y,X2,Y2,X1,Y1,X3,Y3,i).
{ reachable(X1,Y1) } :- reachable(X,Y), point(X1,Y1), neighbor(X2,Y2,X3,Y3),
                        cell(X2,Y2), cell(X3,Y3), cell(X,Y), cell(X1,Y1),
                        relr3(X,Y,X2,Y2,X1,Y1,X3,Y3,iii).
                    :- point(X,Y), cell(X,Y), cell(X1,Y1),
                       not reachable(X1,Y1), relr4(X,Y,X1,Y1,i).
neighbor(X,Y,X1,Y1) :- dist1(X,X1), cell(X,Y), cell(X1,Y1), relr5(Y,Y1,i).
{ neighbor(X,Y,X1,Y1) } :- dist1(X,X1), cell(X,Y), cell(X1,Y1), relr5(Y,Y1,iii).
neighbor(X,Y,X1,Y1) :- dist1(Y,Y1), cell(X,Y), cell(X1,Y1), relr6(X,X1,i).
{ neighbor(X,Y,X1,Y1) } :- dist1(Y,Y1), cell(X,Y), cell(X1,Y1), relr6(X,X1,iii).
```

Fig. 16. Non-ground abstract program constructed by mDASPAR.

atoms in the abstract program are named w.r.t. the rule number. For example, the constraint (75) is numbered r_4, and the standardization creates the relations $X = X_1$ and $Y = Y_1$; in the abstraction the joint relation type atom becomes $\tau_{\mathrm{I}}^{=,=}(X, Y, X_1, Y_1)$ for type I. The abstracted constraint containing the type III relation atom, i.e., $\tau_{\mathrm{III}}^{=,=}(X, Y, X_1, Y_1)$, in its body is unsatisfiable and gets omitted in the abstraction (similar to omitting (14) in Example 3.13).

Furthermore, standardizing apart the variables of the negative literal in (75) relaxes its aim of ensuring that all points are reachable to hold only when the abstraction is refined enough to satisfy the relation atom `relr4(X,Y,X1,Y1,i)`. Having (75) without standardization would ensure it is satisfied in coarser abstractions. We standardized apart the variables of the negative literal as well to obtain more fine-grained abstractions that distinguish the original cells to reach a concrete solution. This makes it easier to visualize the resulting abstractions and understand solutions obtained.

6.2.3. Implementation aspects of mDASPAR

Two-phase debugging. The multi-dimensionality of the domain mapping gives rise to many possible causes of spuriousness. Debugging the non-ground spuriousness by searching for the answer set with smallest number of *ab* atoms can become more difficult. To handle this, we implemented a two-phase debugging approach. In phase 1, the debugging program Π_{debug} is created by modifying the debugging atoms $ab\_deact, ab\_deactCons$ of Definition 4.5 to have only the rule name as arguments. We denote this program by Π_{debug_0}. This then results in an easier computation of an answer set with minimal *ab* atoms. In phase 2 a new program Π_{debug} is created according to the original definition, but the *ab* atoms are only created for the rule names or atoms occurring in the *ab* atoms of I. This way, the search for an optimal answer set focuses on the trouble-making rules/atoms.

Steer debugging towards constraints. In the problems we focus on, the constraints in the program cause to have unsatisfiability or to obtain a particular solution for a given instance. In order to help with reaching abstractions where the relevant constraints are distinguished, we assign less cost to obtaining answer sets with $ab\_deactCons$ atoms in the optimal answer set search during debugging.

Getting hints. Since the refinement of a region means to split it into four subregions, we only need to get the hint of which region to refine. This is different from the hints obtained for DASPAR, as there a decision for refinement relies on the domain elements occurring in the debugging atoms. We alter the *refine* atoms to get the information of which abstract domain occurs as a reason for spuriousness.

Modular concreteness checking. In some cases, even the two-phase checking may not help with easily finding the optimal answer set during the debugging step as the original domain is large or many atoms cause to consider many possible concretizations. We thus considered two orthogonal approaches:

(1) For programs that are modular and contain a clear order on the atoms (e.g. in a plan), the checking is done incrementally over the approach, similar in spirit to [53] which builds on the concept of modules [94].
(2) Using a hierarchy of abstractions (as possible by Proposition 3.6), the checking is done via incrementally concretizing the abstract domain, following an iterative deepening style (Fig. 17).

The aim of Approach (1) is to avoid checking the whole ordered sequence of atoms (e.g., a plan), and catching the spuriousness in some prefix. Approach (2) is applied to avoid making the concreteness check directly at the original domain. If the abstract answer set is spurious, this may be detected in the partially concretized domain. We then check correctness of $\hat{I}$ on the abstract level m_i using Π^{m_i}. If $\hat{I}$ is concrete w.r.t. the partially concretized abstraction, the concretization is

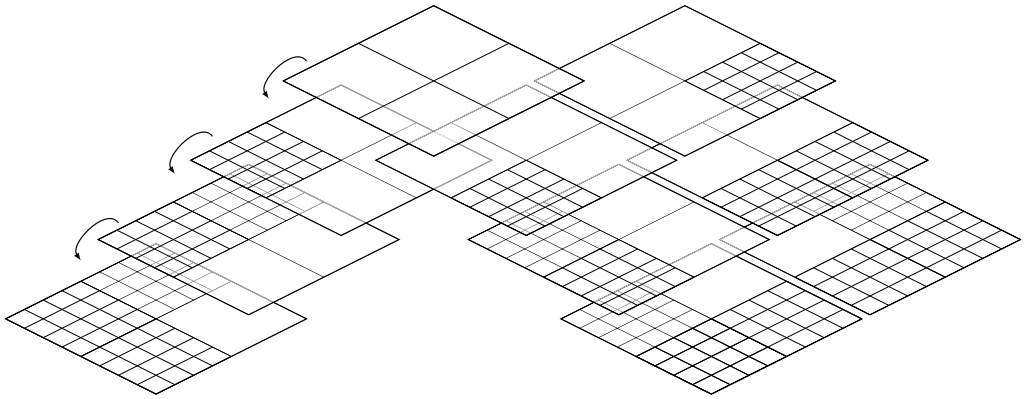


Fig. 17. Step-wise partial concretization of a grid-cell abstraction.

increased for redoing the check. However, if spuriousness is detected, the mapping is refined and the partial concretization continues from the updated mapping.

Further details on the implementation can be found in Appendix B.4.

6.3. Evaluation: obtaining abstract solutions

The main aim of our evaluation was to see whether the domain abstraction and refinement method from above can find automatically non-trivial domain abstractions that yield concrete answer sets. We also wanted to observe the effect of variants of picking abstract answer sets (Section 6.1) or making refinement decisions (Section 4).

6.3.1. Experiments

We used DASPAR v0.2 for the experiments, which employs the sophisticated debugging program for concreteness checking.[5] The variations we considered are as follows.

- When computing abstract answer sets, we either **(s)** pick a single abstract answer set or **(div)** pick a (diverse) set of answer sets w.r.t. the focus atoms.
- Deciding on a refinement is either done **(v1)** by assigning costs to possible refinements and picking the one with smallest cost or **(v2)** by using the hints obtained from the debugging atoms while checking. For **(v2)**, it is ensured that the distinct domain elements in the picked debugging atom do not occur in the same cluster.

We conducted experiments on two benchmark problems from ASP competitions, viz. graph coloring and disjunctive scheduling. For graph coloring, we randomly generated 20 graphs on 10 nodes with edge probability $0.1, 0.2, \ldots, 0.5$ each; out of the 100 graphs, 74 were 3-colorable. We used two different graph coloring encodings shown in Fig. 18, to see their effect in the resulting abstractions. In the first encoding GC_{enc1} (Fig. 18a), a color assignment to each node is guessed as (76)-(78) with the common approach of using default negation, the auxiliary atom $hasEdgeTo(X, C)$ shows which colors C the node X has as its neighbors. The second encoding GC_{enc2} (Fig. 18b) uses a choice rule (83) to guess an assignment and then ensures with (84) that a node is not assigned more than one color. The rules (85)-(86) are an alternative way of writing the rule $\bot \leftarrow chosenColor(X_1, C), chosenColor(X_2, C),\ edge(X_1, X_2), X_1 < X_2$. so that when the variables are standardized apart for the sort *node*, fewer relation atoms occur in one rule. Also notice that GC_{enc2} imposes an order relation among the nodes, to reduce duplications of the constraints.

For disjunctive scheduling, for each $t \in \{10, 20, 30\}$, we generated 20 instances with 5 tasks over time $\{1, \ldots, t\}$. We used the encoding[6] from ASP Competition 2011 and precomputed the deterministic part (i.e., not involved in unstratified negation resp. guesses) of the program, so that they are lifted to the abstract program without introducing (unnecessary) nondeterminism (see Appendix B.2). The initial abstraction mapping is the single-cluster abstraction, i.e., clustering all nodes into one for graph coloring and all time points into one for disjunctive scheduling.

In the experiments, we use the lifted relation approach. While in existential abstraction on the relations (Section 5.1) there are fewer relation types to consider, we observed no improvement in the computation effort, since type III for abstract relations is a combination of type III and IV of lifted relations. Thus, it does not make a difference which relation type approach is used in the experiments.

[5] Results for version v0.1 were reported in [113].

[6] www.mat.unical.it/aspcomp2011/files/DisjunctiveScheduling/disjunctive_scheduling.enc.asp.

$$chosenColor(X, r) \leftarrow not\ chosenColor(X, g), not\ chosenColor(X, y), node(X). \quad (76)$$

$$chosenColor(X, g) \leftarrow not\ chosenColor(X, r), not\ chosenColor(X, y), node(X). \quad (77)$$

$$chosenColor(X, y) \leftarrow not\ chosenColor(X, g), not\ chosenColor(X, r), node(X). \quad (78)$$

$$hasEdgeTo(X, C) \leftarrow edge(X, Y), chosenColor(Y, C). \quad (79)$$

$$\bot \leftarrow hasEdgeTo(X, C), chosenColor(X, C). \quad (80)$$

$$colored(X) \leftarrow chosenColor(X, C). \quad (81)$$

$$\bot \leftarrow node(X), not\ colored(X). \quad (82)$$

(a) GC_{enc1}

$$\{chosenColor(X, C)\} \leftarrow node(X), color(C). \quad (83)$$

$$\bot \leftarrow chosenColor(X, C_1), chosenColor(X, C_2), C_1 \neq C_2. \quad (84)$$

$$adj(X, Y) \leftarrow edge(X, Y), X < Y. \quad (85)$$

$$\bot \leftarrow adj(X, Y), chosenColor(X, C), chosenColor(Y, C). \quad (86)$$

$$colored(X) \leftarrow chosenColor(X, C). \quad (87)$$

$$\bot \leftarrow node(X), not\ colored(X). \quad (88)$$

(b) GC_{enc2}

Fig. 18. Two encodings of the Graph Coloring problem: GC_{enc1} and GC_{enc2}.

Table 4
Experimental results for graph coloring.

		full				projected			
		(s)		(div)		(s)		(div)	
		(v1)	(v2)	(v1)	(v2)	(v1)	(v2)	(v1)	(v2)
GC_{enc1}	number of steps	7.38	7.83	7.04	7.69	5.24	6.48	4.83	6.14
	abstraction domain size	8.38	8.84	8.04	8.69	6.24	7.48	5.83	7.14
	faithful abstraction domain size	6.84	8.04	6.12	7.51	6.02	5.71	5.65	5.82
	trivial abstractions (*id*)	13%	23%	4%	12%	2%	1%	2%	2%
	faithful & non-*id* abstractions	30%	32%	29%	27%	56%	61%	50%	47%
	non-faithful abstractions	57%	45%	67%	61%	42%	38%	48%	51%
GC_{enc2}	number of steps	7.01	6.40	6.56	6.37	3.53	3.76	3.40	3.52
	abstraction domain size	8.01	8.64	7.56	8.29	4.53	6.73	4.40	6.36
	faithful abstraction domain size	8.88	8.62	7.97	8.66	4.86	5.44	4.75	5.72
	trivial abstractions (*id*)	19%	13%	5%	13%	3%	2%	3%	2%
	faithful & non-*id* abstractions	22%	24%	25%	22%	54%	59%	54%	48%
	non-faithful abstractions	59%	63%	70%	65%	43%	39%	43%	50%

6.3.2. Results

We report the average results over 10 runs for each variation. To ease presentation, we discuss the results for each benchmark separately by concentrating on the different observations made throughout the experimental evaluation.

Graph coloring. The evaluation results of the obtained abstractions are presented in Table 4. The first two rows show the average number of refinement steps and the average domain size (i.e., the number of clusters) of the resulting abstractions. The best abstraction (i.e., with smallest domain size) found for each instance in the runs is further checked for faithfulness, to observe whether the corresponding abstract program only contains concrete answer sets. The domain size of the faithful abstractions is shown in the third row. The frequencies of the abstractions that are trivial (thus faithful), non-trivial and faithful, and non-faithful are shown in the last three rows.

The left column shows the results of full concreteness checking with different variations. We can observe that deciding on a refinement based on single abstract answer set **(s)** results in finer abstractions (i.e., with larger domain size) than on diverse set of abstract answer sets **(div)**. The number of trivial abstractions obtained is also smaller for **(div)** (better decisions are made) and the chance of encountering a concrete abstract answer set is larger. The latter causes to obtain more non-faithful abstractions, as then no refinement to an abstraction with less spurious answer sets is made. As for using **(v2)**, i.e., to decide on refinements, we can observe that this is not better for obtaining coarser abstractions than the minimal

Table 5
Experimental results for scheduling.

time		(s)			(div)		
		(v1)	(v2)	(v2')	(v1)	(v2)	(v2')
$t = 10$	number of steps	7.22	4.81	3.56	6.04	4.81	3.54
	abstract domain size	8.22	8.48	8.46	7.04	8.38	8.35
	calls abstract program	41.35	5.81	4.56	40.72	5.81	4.54
	calls debugging program	40.90	5.36	4.11	56.30	6.87	5.44
$t = 20$	number of steps	14.71	7.65	5.47	12.00	7.53	5.33
	abstract domain size	15.71	14.16	14.12	13.00	14.16	13.81
	calls abstract program	168.48	8.65	6.47	157.41	8.53	6.33
	calls debugging program	168.28	8.45	6.27	244.45	12.08	8.74
$t = 30$	number of steps	22.82	9.57	7.76	20.57	9.56	7.68
	abstract domain size	23.82	19.02	19.12	21.57	19.07	18.68
	calls abstract program	391.88	10.57	8.76	366.09	10.56	8.68
	calls debugging program	391.43	10.12	8.31	580.23	14.59	12.24

cost method **(v1)** in general; it also yields more trivial abstractions than **(v1)**, as splitting the domain repeatedly to break up clusters of certain abstract elements quickly ends up with the original domain.

The right column shows the results for a projected notion of concreteness that limits checking to a set of relevant atoms; for this, we picked the nodes 1,2,3 and their assigned colors. As expected, a concrete abstract answer set is encountered in much coarser abstractions, as the colors assigned to the other nodes do not matter. In case of projection, the trivial abstraction is reached much less often; moreover, more non-trivial faithful abstractions are reached. This is beneficial, as the computed abstractions can be used to obtain all (concrete) solutions over the nodes focused on.

The main difference of the various encodings is the size of the achieved abstract domains. GC_{enc2} requires fewer refinement steps to achieve an abstraction with a concrete solution than GC_{enc1}, as the need to preserve the node ordering leaves fewer refinement possibilities. On average, the resulting abstractions are coarser than by GC_{enc1} while the domain sizes of the faithful abstractions are larger. This may be due to the choice rule in GC_{enc2}, causing spurious answer sets that must treated by further refinement steps.

Disjunctive scheduling. We compared the effects of the variations for the resulting abstractions and the calls to the ASP solver to obtain an abstract answer set, respectively, to check concreteness with debugging; Table 5 shows the collected results. For the refinement search, we considered besides **(v1)** and **(v2)** the variant **(v2')** of **(v2)** where each abstract element in the obtained debugging atom is mapped to a singleton cluster in the refinement.

As expected, the minimal cost method **(v1)** causes much more calls to the ASP solver, as the cost for each possible refinement must be computed. While it achieves coarser abstractions in half of the cases, the large number of calls is a clear disadvantage. For example, for the case $t = 20$, **(v1)** achieves with **(div)** on average an abstract domain of 13.00 clusters with 400 calls to the ASP solver, while **(v2')** achieves an average of 13.81 clusters with only around 15 calls.

For the instances with $t = 20$, refinement through hints **(v2)** achieves coarser abstractions than **(v1)** when single abstract answer sets are picked. Here hints guide the refinement much better than the cost from a single abstract answer set. For the cost from a diverse set of abstract answer sets, significantly coarser abstractions are achieved. Looking at $t = 30$, we can observe that the cost approach **(v1)** results in much finer abstractions than the hint based approaches **(v2)** and **(v2′)**, which provide better guidance. This shows that a local search over the 1-step refinements does not always yield the best outcome, and it is moreover also more expensive.

We can also observe that **(v2′)** mostly achieves coarser abstractions than **(v2)**; immediately singling out the domain elements connected with the spuriousness helps. It also needs the smallest number of refinement steps compared to other approaches, as it reaches a concrete solution much faster with the refinement decisions.

The results show that with larger domains, the effect of the abstraction can be seen much better; e.g., the best abstract domain size reached for $t = 10$ on average is 70.4% ($= 7.04/10$) of the original domain size, while for $t = 30$ it shrinks to 62% ($= 18.68/30$).

Summary. The results show that with domain abstraction it is possible to achieve concrete solutions while abstracting over some of the details of the program. Reaching faithful abstractions is desired; however it does not occur often, unless a projected concreteness check is considered that only distinguishes the details relevant for a solution of the problem. Obtaining hints from a set of abstract spurious answer sets instead from a single such answer set results in better decisions and thus coarser abstractions.

6.4. Evaluation: unsolvable problem instances in grid-cells

We investigated obtaining explanations of unsatisfiable grid-cell problems by achieving an abstraction over the instance to focus on the troubling area. We considered the following benchmark problems:

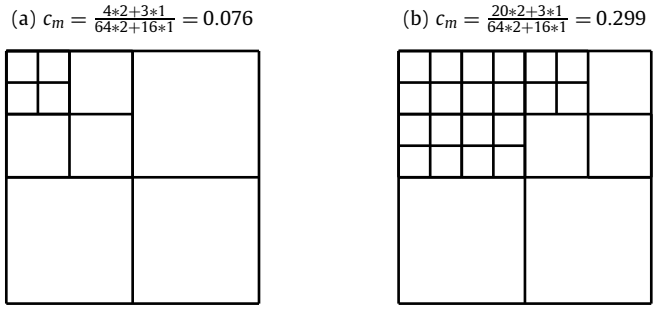


Fig. 19. Measure for quality of a quad-tree abstraction.

- *Reachability (R)*: This problem needs the neighboring cell information and can be encoded without introducing guesses. We check whether every cell is reachable; unsatisfiability is due to the layout of the obstacles.
- *Knight's Tour (KT)*: This problem is on finding a tour on which a knight visits each square of a board once and returns to the starting point. It is commonly used in ASP Competitions, with possible addition of forbidden cells. Unsolvability is due to forbidden cells that prevent the knight from moving. In ASP competitions, this problem is encoded by guessing a set of $move(X_1, Y_1, X_2, Y_2)$ atoms and ensuring that each cell has only one incoming and one outgoing movement.[7] There is no *time* sort (as in planning) which would describe an order.
- *Visitall*: We extended the planning problem of visiting every cell (without revisiting a cell) with obstacles. This problem needs the neighboring cell information and can be encoded in two forms;
  (V) as a planning problem, in order to find a sequence of actions that visits every cell, or
  ($\mathbf{V_{KT}}$) as a combinatorial problem similar to the Knight's Tour encoding.
  To allow for shorter plans, we encoded **(V)** using $go(X, Y, T)$ actions that can move horizontally/vertically to a cell X, Y (without passing through obstacles) and the passed cells become visited; we set a limit of 30 time steps.
- *Sudoku (S)*: This problem has also been used in ASP competitions.[8] Its encoding consists of a guess of numbers in the cells combined with simple constraints such as one symbol per column, one symbol per subregion etc. The unsolvability occurs due to violation of these constraints.

We generated 10 unsatisfiable instances complying to the following properties so that the unsolvability can be explained by focusing on a troubling area[9]:

- In Reachability instances, a group of neighboring cells is unreachable due to the obstacles surrounding them.
- For Knight's Tour instances, one or two cells are picked to have only one valid movement to an obstacle-free cell. This way, these cells and the obstacles that do not allow the valid movements become a reason for unsolvability.
- The Visitall instances consist of either two dead-end cells or areas with only one cell passage, so that one is forced to pass some cells more than once, which is not allowed.
- For Sudoku, we generated a layout of numbers that force to violate the constraints when solving the problem.

Measuring abstraction quality We consider a *quality measure* of the quad-tree abstraction by normalizing the number of abstract regions of certain size and their level in the quad-tree. The cost of a mapping m over an $n \times n$ grid is

$$c(m) = \sum_{i=0}^{\ell} r_{2^i}(m)(\ell - i) \,/\, \sum_{i=0}^{\ell} n^2 2^{-i^2}(\ell - i),$$

where $\ell = \log_2(n) - 1$ is the level, $r_{2^i}(m)$ is the number of abstract regions of size $2^i \times 2^i$ in m, and $n^2 2^{-i^2}$ is the number of abstract regions of size $2^i \times 2^i$ in the $n \times n$-sized cell. The factor $\ell - i$ is a weight that gives higher cost to abstractions with more low-level regions. An abstraction mapping with the smaller cost, i.e., intuitively smaller level of detail, is considered to be of *better quality*.

Fig. 19 shows measures of two abstraction mappings. The abstraction in Fig. 19a is coarser than the one in Fig. 19b, and this is reflected in the computed measures. Assigning more weight to having coarser regions would stress the importance of having a coarse abstraction even more. The computation of the measure is purely structural and domain-independent. Other measures can be defined that are dependent on the domain which considers further aspects, e.g., such that an abstraction that singles out the smallest number of cells with obstacles is preferred.

[7] www.mat.unical.it/aspcomp2013/KnightTour.

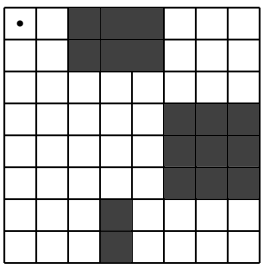
[8] dtai.cs.kuleuven.be/events/ASP-competition/Benchmarks/Sudoku.shtml.

[9] All benchmark instances and encodings as well as user explanations for Visitall and Reachability are available at www.kr.tuwien.ac.at/research/systems/abstraction/mdaspar_material.zip.

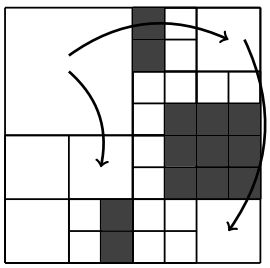
Table 6
Evaluation results of applying different debugging approaches.

	debugging type	average steps	average cost	minimum steps	minimum cost	best steps	best cost
R	default	5.4	0.227	5.4	0.227	5.0	0.208
	2-phase	5.5	0.233	5.3	0.222		
S	default	6.5	0.696	5.1	0.550	3.2	0.371
	2-phase	4.3	0.476	3.4	0.391		
KT	2-phase	14.3	0.643	10.4	0.460	5.6	0.245
	grid-inc	10.1	0.442	6.3	0.277		
V	2-phase[a]	16.2	0.708	13.9	0.608	8.7	0.360
	time-inc	16.3	0.712	13.5	0.569		
V_{KT}	2-phase	15.7	0.693	13.0	0.572	7.6	0.317
	grid-inc	13.0	0.569	10.3	0.449		

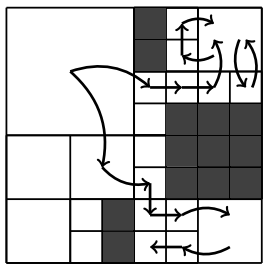
[a] A total of 16 runs could not be completed due to memory errors. The results are computed among the runs that have been completed.



(a) inst. #10



(b) V - spurious action transitions



(c) V_{KT} - separate action sequences that visit the cells reachable with single-cell passages

Fig. 20. Spurious plans in abstractions that distinguish the single-cell passages.

6.4.1. Effects of different debugging approaches

We compared different debugging approaches from Section 6.2.3 to observe their effects on the resulting abstractions and the taken refinement steps. Due to their encodings and constraints, the Knight's Tour and Visitall problems are the challenging ones. To observe whether an incremental checking could help in deciding on a refinement and achieve better abstractions, we applied for KT and V_{KT} partial concretization and for V incremental *time* checking. To evaluate how far a resulting abstraction is from the *best possible* abstraction showing unsolvability, we also checked whether a coarser abstraction with this property exists.

Table 6 shows the main evaluation results. We compare different debugging approaches in terms of the average refinement steps and average costs of the resulting abstractions over 10 runs, and also on the best outcome obtained (with minimum refinement steps and minimum mapping cost). The two right-most columns concern the existence of a coarser abstraction for best outcome obtained. The time to find an optimal solution in the debugging step was limited to 50 seconds. If exceeded, the refinement is decided on the basis of suboptimal analyses by considering the optimal debugging solution that could be computed within the time limit.

For Reachability and Sudoku, we observe that abstractions close to the best possible ones can be obtained. Better abstractions were obtained with 2-phase debugging in these cases (the majority with a clear margin for S), as after the first step the focus was on the right part of the abstraction. For Knight's Tour and Visitall, we observe that incremental checking can obtain better abstractions. This is because for 2-phase debugging, the programs mostly had due to timeouts to decide on suboptimal concreteness checking outputs. Moreover, for the V encoding 2-phase debugging caused memory outages (limit 500 MB) on some runs for some instances, thus not all 10 runs could be completed.

We can also see a difference of the resulting abstractions for the two encodings of Visitall. The planning encoding achieves unsatisfiability with less coarse abstractions. Guesses of spurious sequences of actions in the abstraction cause the debugging to decide on refinements that avoid these sequences. The focus moves towards the unsolvability when the abstract action sequence is not executable due to an obstacle. In some instances where the reason for unsolvability is not easily caught by having two dead-ends, focusing on the existence of obstacles does not achieve unsatisfiability: the abstract encoding manages to find a plan passing through different sized regions by avoiding the constraints due to uncertainty. For these instances, the abstraction needs to be fine-grained enough to get rid of most of the uncertainty.

Fig. 20a shows an example of such an instance. An abstraction that distinguishes the one-passage-entries and the obstacles that surround the cells cannot achieve unsatisfiability for encoding V. Fig. 20b shows some spurious action transitions

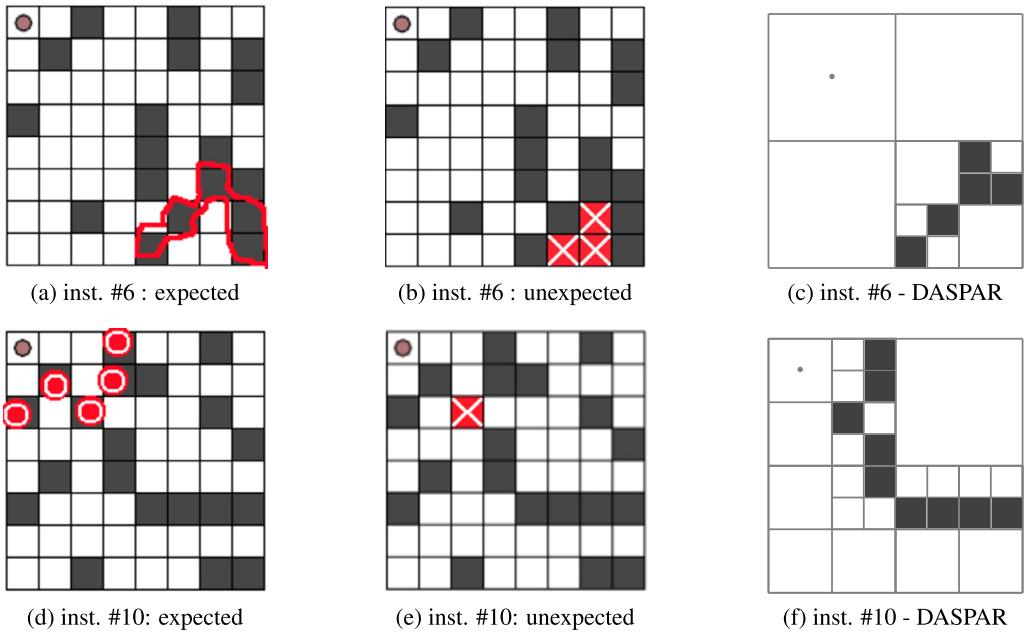


Fig. 21. Explanatory abstractions for unsatisfiable Reachability instances.

that are determined in a plan found with the V encoding among regions by avoiding the constraints due to uncertainty. Unsatisfiability cannot be achieved for V_{KT} as well. This is due to guess of *move* atoms which achieve that every cell is visited, but lack a corresponding original order of movements. Fig. 20c shows the spurious order of movements that is split to visit each cell that is only reachable through a one-passage-entry. If the abstraction is refined to distinguish the cells in the respective corners, then unsatisfiability is realized.

6.4.2. User study on unsatisfiability explanations

We were interested in checking whether the obtained abstractions match the intuition behind a human explanation. For Reachability and Visitall, finding the reason for unsolvability of an instance is possible by looking at the obstacle layout. Thus, we conducted a user study for these problems in order to obtain the regions that humans focus on to realize the unsolvability of the problem instance.

As participants, we had ten PhD students of Computer Science at TU Wien. We asked them to mark the area which shows the reason for having unreachable cells in the Reachability instances and the reason for not finding a solution that visits all the cells in the Visitall instances; multiple reasons are to mark with different colors. Explanations for 10 instances of each problem were collected.[9] We discuss the results for both problems by showing two of the responses (expected and unexpected) and the best abstraction obtained from mDASPAR when starting with the initial mapping.

Reachability. The expected explanations (e.g., Figs. 21a and 21d) focus on the obstacles that surround the unreachable cells, as they prevent them from being reachable. When their respective abstraction mappings are given to mDASPAR, the constructed abstract program is also unsatisfiable. The explanation in Fig. 21b puts the focus on the unreachable cells themselves, and Fig. 21e distinguishes a particular obstacle as a reason. The mark in Fig. 21e is actually a possible solution to the unreachability of the cells, since removing the marked obstacle makes all the cells reachable. When the respective abstraction mappings are given to mDASPAR, it needs to refine the abstraction further to distinguish more obstacles and to realize the unsatisfiability.

In ASP, checking whether all cells are reachable is straightforward, without introducing guesses. This is also observed to be helpful for mDASPAR, as most of the resulting abstractions were similar to the gathered answers. Since in the initial abstraction, the abstract program only knows that the agent is located in the upper-left abstract region, in instance #10, mDASPAR follows a different path in refining the abstraction, and reaches the abstraction shown in Fig. 21f. Although different from the one by the users, it also shows a reason for having unreachable cells. Humans use the implicit knowledge that the agent is located in the upper-left corner in order to determine the reason for unreachability of the cells, and thus focus on a different area than mDASPAR. Such an abstraction can also be achieved with the method, by influencing the refinement decisions towards singling out the initial location of the agent.

The abstractions achieved by mDASPAR are more general as the precise initial location of the agent is immaterial to distinguish the unreachable cells: it can be in any of the cells mapped to the respective abstract region. The precise obstacle layout in the abstracted regions also plays no role in determining the unreachability of the distinguished cells.

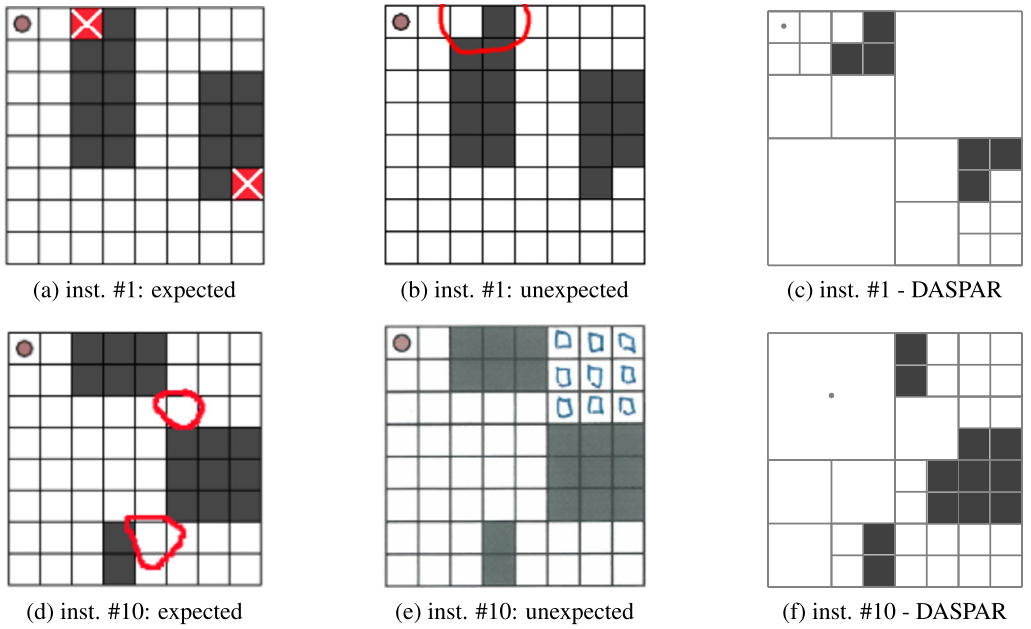


Fig. 22. Explanatory abstractions for unsatisfiable Visitall instances.

Visitall. Most of the users picked two dead-end cells in the instances (if such occur) as an explanation for unsatisfiability, instead of the obstacles surrounding these cells (see Fig. 22a), which are the actual cause for them being dead-end cells. Even with abstraction mappings that also distinguish the surrounding obstacles of these dead-end cells, the corresponding abstract program still remains satisfiable. Then mDASPAR needs to further refine the abstraction to distinguish the neighboring cells (as in Fig. 22c) and to realize that it can only pass through one grid-cell when reaching the dead-end cells, and thus achieve unsatisfiability. Unexpectedly, some users marked only one of the dead-end cells as an explanation (Fig. 22b), which is actually focusing on a possible solution to the unsolvability, since if the marked area was not a dead-end, all cells could have been visited.

Some instances do not contain two dead-end cells, but single-cell passages to some regions. Fig. 22d shows an entry that distinguishes these passages, while again focusing only on the cells themselves. For these instances, the results of mDASPAR are quite different. A discussion on why having an abstraction that distinguishes the one-passage-entries and the obstacles that surround the cells cannot achieve unsatisfiability was given for Fig. 20 in Section 6.4.1. Fig. 22f shows the best abstraction achieved for V_{KT}. Instead of focusing on the passages, it distinguishes all cells in the one-passage-entry regions to realize that a desired action sequence that will manage to visit all cells without revisiting one cannot be found.

The generality of the achieved abstractions is also here observable: the precise agent position and obstacle layout in the abstracted areas do not change the unsatisfiability result as not all cells in the distinguished parts can be visited.

Observations The abstraction method can demonstrate the capability of human-like focus on certain parts of the grid to show the unsolvability reason. However, humans implicitly also use their background knowledge and do not need to explicitly state the relations among the objects. Empowering machines with such capabilities remains a challenge. The study also showed a difference in understanding the meaning of "explanation". For some study participants, showing how to get rid of unsolvability was also seen as an explanation. This discrepancy shows that one needs to clearly specify what is wanted (e.g., "mark only the obstacles that cause to have unreachable cells"), to achieve less variety of results.

7. Discussion

In this section, we first discuss a possible way to achieve abstraction over predicates using our method, and we then focus on the ASP planning use case in order to highlight the potential of domain abstraction in finding the essence of planning problems. After that, we discuss related work in the literature.

7.1. Predicate abstraction

Predicate abstraction in ASP would introduce literals involving new predicates that describe an abstraction of original literals, and rewriting the program to mention only the new literals. Naively replacing literals with the abstract ones would not always achieve an over-approximation.

Example 7.1 *(Example 3.1 ctd).* Consider a predicate abstraction that maps the atoms $a(X)$ and $d(X)$ to $ad(X)$, i.e., if $a(d)$ or $d(d)$ holds true for some $d \in D$, then $ad(d)$ holds true. When we replace the a- and d-atoms in Example 3.1, we obtain the following program:

$$b(X,Y) \leftarrow ad(X), ad(Y).$$
$$e(X) \leftarrow c(X), ad(Y), X \neq Y.$$
$$\bot \leftarrow b(X,Y), e(X).$$
$$ad(1).\ ad(3).\ c(2).\ ad(5).$$

However, the answer set of the program contains $\{b(1,1), b(3,1), \dots\}$ and it does not match the original answer set.

A simple way to achieve predicate abstraction is via domain abstraction after reification of predicates of the original program. For example, $p(X,Y)$ is written to $x(p,X,Y)$ and a sort of predicate names (viewed as constants) is introduced. By standardizing apart the variables, predicate names can be clustered via the built-in relations.

Example 7.2 *(ctd).* Rewriting the atoms $a(X)$ and $d(X)$ to $x(a,X)$ and $x(d,X)$, respectively, yields the program

$$b(X,Y) \leftarrow x(P_1,X), x(P_2,Y), P_1 = a, P_2 = d, pred(P_1), pred(P_2).$$
$$e(X) \leftarrow c(X), x(P,Y), P = a, pred(P), X \neq Y.$$
$$\bot \leftarrow b(X,Y), e(X).$$
$$x(a,1).\ x(a,3).\ c(2).\ x(d,5).$$
$$pred(a).\ pred(d).$$

Then an abstraction m over the sort $pred$ such as $\{a,d\}\mapsto ad$ can be applied.

This approach works for predicate abstraction where the corresponding literals have arguments from the same sort in the same argument position. In case a literal has fewer arguments, dummy values can be used to fill in the remaining argument positions.

Another way to achieve predicate abstraction is by following the motivation behind existential abstraction of the relations. The idea is to introduce a new set of predicates along with their relation types according to the abstraction; then the abstract rules will be formed for all combinations of the abstraction types in the bodies, where choice is added to the head unless all are type I.

Example 7.3 *(ctd).* Similar to Example 7.1, the abstract predicate name ad is introduced with the relation type τ^{ad}_{III}. Note that the arguments of the literal are not important, as the abstraction is over the predicate name and not over domain elements. The abstract program is then as follows:

$$\{b(X,Y)\} \leftarrow ad(X), ad(Y), \tau^{ad}_{\text{III}}.$$
$$\{e(X)\} \leftarrow c(X), ad(Y), \tau^{ad}_{\text{III}}, X \neq Y.$$
$$\bot \leftarrow b(X,Y), e(X).$$
$$ad(1).\ ad(3).\ c(2).\ ad(5).$$

We remark that this approach is similar to using the rewriting of the original program with reification of predicates, and applying existential abstraction on the relations.

7.2. Use case: abstraction in ASP planning

Domain abstraction gives us the possibility to adjust the granularity of a problem towards the relevant details. By achieving abstract answer sets that are concrete and thus catch all the relevant details, it also allows for problem solving over abstract notions, which can be useful in a wide range of applications. We discuss here the possible use of domain abstraction in ASP planning, in particular, in understanding planning problems expressed in ASP by abstracting over the unnecessary details. Another use case about policy refutation is described in Appendix C.

Planning problems in ASP are represented by using a `time` sort to describe the sequence of states and the changes according to actions taken [77]. There are usually two types of objects, represented with different sort types:

- objects on which the actions have a direct effect, e.g., the blocks in the blocksworld which can be moved, and

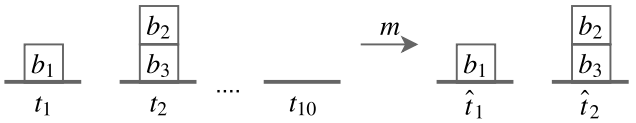


Fig. 23. Initial state of a blocksworld with multiple tables (concrete $\xrightarrow{m}$ abstract).

- other objects unaffected by the actions but involved in the decision making, e.g., the table in the blocksworld onto which blocks can be moved.

We discuss how to use domain abstraction over these objects such that we can talk about abstract states and plans.

Describing actions in ASP First, we emphasize on the blocksworld example the different ways of expressing planning problems in ASP. The effects of moving a block on top of another block are expressible by the following rules:

$$onB(B, B_1, T+1) \leftarrow moveToBlock(B, B_1, T). \tag{89}$$

$$\neg onB(B, B_2, T) \leftarrow onB(B, B_1, T), B_1 \neq B_2. \tag{90}$$

where (89) models the direct and (90) the indirect effect. Alternatively, all effects are expressed as direct effects by

$$onB(B, B_1, T+1) \leftarrow moveToBlock(B, B_1, T). \tag{91}$$

$$\neg onB(B, B_2, T) \leftarrow moveToBlock(B, B_1, T), B_1 \neq B_2. \tag{92}$$

The preconditions of an action can either be described through constraints, or as a condition for an action to become applicable. For example, that a block cannot sit on a block with a smaller name can be expressed as a constraint

$$\bot \leftarrow onB(B, B_1, T), B_1 \leq B.$$

Alternatively, the respective action can be forbidden if the condition is not satisfied using the following rules:

$$\bot \leftarrow moveToBlock(B, B_1, T), not\ precondmtb(B, B_1, T).$$

$$precondmtb(B, B_1, T) \leftarrow B < B_1, block(B), block(B_1).$$

Note that the alternative version is much closer to the PDDL-style encoding. The law of inertia is described by the rule

$$onB(B, B_1, T+1) \leftarrow onB(B, B_1, T), not\ \neg onB(B, B_1, T+1).$$

7.2.1. Abstracting over irrelevant details

We first show the possibility of abstraction over the details of the objects that are indirectly affected by the actions. For demonstration, we consider two extensions of well-known planning domains.

- *Multi-table blocksworld (MTB)*: here blocks can be moved onto one of multiple tables (where each table can hold multiple blocks); a plan is needed that piles the blocks up on a given specific table.
- *Package delivery with checkpoints (PDC)*: packages must be carried from an initial to a goal location, while passing through a checkpoint reachable from the initial location.

Multi-table blocksworld. Fig. 23 illustrates an instance of MTB where the blocks must be piled up on table t_1 such that b_1 is above b_2 and b_2 is above b_3. Here reaching the goal state does not depend on the concrete tables to which blocks are moved before moving them to the goal table. However, when computing a plan based on the original program, the planner has to consider all possible movements.

Fig. 24 shows a (natural) encoding of the problem with the actions $moveToT(B, Ta, T)$ and $moveToB(B, B', T)$ for moving block B onto table Ta and onto block B', resp., at time T. Consider the initial state shown in Fig. 23:

$$onT(b_1, t_1), onB(b_2, b_3), onT(b_3, t_2), chosenTable(t_1).$$

After ensuring that all variables are guarded by domain predicates and those related with the *table* sort are standardized apart, we run DASPAR with the initial mapping $\{\{t_1, \ldots, t_n\} \mapsto \hat{t}\}$. The abstraction obtained is shown in Fig. 23; it singles out the chosen table $\hat{t}_1$ and clusters all others into $\hat{t}_2$. We then can compute a concrete abstract answer set

$$\{moveToT(b_2, \hat{t}_2, 0), moveToT(b_3, \hat{t}_1, 1), moveToB(b_2, b_3, 2), moveToB(b_1, b_2, 3)\}.$$

% action choice
$\{moveToB(B, B_1, T) : bl(B), bl(B1); moveToT(B, L, T) : bl(B), tbl(L)\} \leq 1 \leftarrow T < t_{max}.$

% no gaps between moves
$done(T) \leftarrow moveToB(B, B_1, T).$
$done(T) \leftarrow moveToT(B, L, T).$
$\bot \leftarrow done(T{+}1), not\ done(T), T < t_{max}.$
% preconditions
$\bot \leftarrow moveToB(B, B_2, T), onB(B_1, B, T).$
$\bot \leftarrow moveToB(B, B_2, T), onB(B_1, B_2, T).$
$\bot \leftarrow moveToT(B, L, T), onB(B_1, B, T).$
$\bot \leftarrow moveToT(B, L, T), onT(B, L, T).$
% effects
$onB(B, B_1, T{+}1) \leftarrow moveToB(B, B_1, T), T{<}t_{max}.$
$onT(B, L, T{+}1) \leftarrow moveToT(B, L, T), T{<}t_{max}.$
$\neg onB(B, B_2, T) \leftarrow onB(B, B_1, T), B_1{\neq}B_2.$
$\neg onT(B, L, T) \leftarrow onB(B, B_1, T).$
$\neg onB(B, B_1, T) \leftarrow onT(B, L, T).$
$\neg onT(B, L_1, T) \leftarrow onT(B, L_2, T), L_1{\neq}L_2.$

% inertia
$onB(B, B_1, T{+}1) \leftarrow onB(B, B_1, T),$
$\quad not\ \neg onB(B, B_1, T{+}1), T{<}t_{max}.$
$onT(B, L, T{+}1) \leftarrow onT(B, L, T),$
$\quad not\ \neg onT(B, L, T{+}1), T{<}t_{max}.$
% state constraints
$\bot \leftarrow onB(B_1, B, T), onB(B_2, B, T), B_1{\neq}B_2.$
$\bot \leftarrow onB(B, B_1, T), onB(B, B_2, T), B_1{\neq}B_2.$
$\bot \leftarrow onB(B, B_1, T), onT(B, L, T).$
$\bot \leftarrow onB(B, B_1, T), B_1 \leq B.$
$\bot \leftarrow onT(B, L_1, T), onT(B, L_2, T), L_1 \neq L_2.$
% goal constraints
$notblockgoal(T) \leftarrow onT(B, L, T), onT(B_1, L_1, T), B{\neq}B_1.$
$\bot \leftarrow notblockgoal(T), T = t_{max}.$
$\bot \leftarrow not\ notblockgoal(T), onT(B, L, T), not\ chosenTable(L).$

Fig. 24. Encoding for Multi-table Blocksworld.

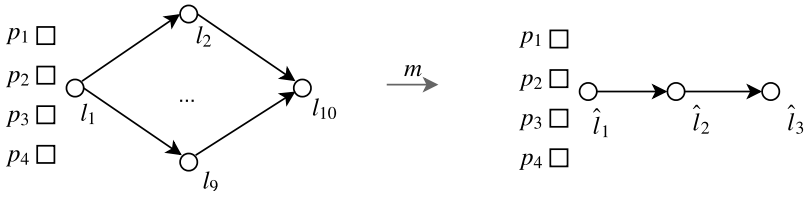


Fig. 25. Initial state of a package delivery with checkpoints (concrete $\xrightarrow{m}$ abstract).

that describes a plan without going into detail on which table the blocks are moved. The abstraction shows that for solving the problem, it is essential to distinguish the picked table from all others; the number of tables is irrelevant. Furthermore, this abstraction is faithful for projection to the actions *moveToB*, *moveToT*.

Package delivery with checkpoints. Fig. 25 illustrates an instance of PDC where the packages in location l_1 must be carried to location l_{10}. As these locations are not directly connected, the truck has to pass through a middle point; through which point the truck passes is immaterial for reaching the goal state.

For this problem, we used the Nomystery encoding from ASPCOMP2015 and altered it to have no fuel computation (shown in Fig. 26). Furthermore, for a $drive(T, L_1, L_2, S)$ action to be possible we added an additional condition that the locations L_1 and L_2 should be connected by an edge, $edge(L_1, L_2)$. Consider the initial state shown in Fig. 25: $\{atT(t, l_1, 0)$, $atP(p_1, l_1, 0)$, $atP(p_2, l_1, 0)$, $atP(p_3, l_1, 0)$, $atP(p_4, l_1, 0)$, $goal(p_1, l_{10})$, $goal(p_2, l_{10})$, $goal(p_3, l_{10})$, $goal(p_4, l_{10})\}$ with the depicted *edge* facts. Running DASPAR with the mapping $\{\{l_1, \ldots, l_{10}\}{\mapsto}\hat{l}\}$ over the sort *location* results in the abstraction mapping $\{\{l_1\}{\mapsto}\hat{l}_1, \{l_2, \ldots, l_9\}{\mapsto}\hat{l}_2, l_{10}{\mapsto}\hat{l}_3\}$ (shown in Fig. 25). With this abstraction, the following concrete abstract answer set is computed:

$$\{load(p_4, t, \hat{l}_1, 1), load(p_3, t, \hat{l}_1, 2), load(p_1, t, \hat{l}_1, 3), load(p_2, t, \hat{l}_1, 4),$$
$$drive(t, \hat{l}_1, \hat{l}_2, 5), drive(t, \hat{l}_2, \hat{l}_3, 6),$$
$$unload(p_3, t, \hat{l}_3, 7), unload(p_1, t, \hat{l}_3, 8), unload(p_4, t, \hat{l}_3, 9), unload(p_2, t, \hat{l}_3, 10)\}.$$

It describes a plan that loads all the packages, moves to the middle cluster location, moves to the goal location, and unloads the packages; the resulting abstraction is faithful for the projection to the actions *load*, *unload*, *drive*.

Domain abstraction simplified details that are unimportant for the essence of whether the plan is feasible. The faithful abstraction gives an understanding of the problem by realizing its neuralgic points. If however there are further constraints over details needed to construct a plan, then faithfulness might not be achievable in a non-trivial abstraction.

7.2.2. Computing abstract plans

Abstracting over the objects directly affected by the actions would empower us to talk about abstract plans. However, in ASP-style encodings, abstracting over the object sort only causes the abstract program to compute plans with the original *time* sort. For example, say in the Package Delivery problem (with no checkpoints and two locations

```
% action choice
{unload(P, T, L, S) : package(P), truck(T), loc(L);
 load(P, T, L, S) : package(P), truck(T), loc(L);
 drive(T, L1, L2, S) : edge(L1, L2), loc(L1), loc(L2), truck(T)} ≤ 1 ← step(S), S > 0.

% no gaps between moves
done(S) ← unload(P, T, L, S).
done(S) ← load(P, T, L, S).
done(S) ← drive(T, L1, L2, S).
⊥ ← done(S+1), not done(S).
% effects
atP(P, L, S) ← unload(P, T, L, S).
¬in(P, T, S) ← unload(P, T, L, S).
¬atP(P, L, S) ← load(P, T, L, S).
in(P, T, S) ← load(P, T, L, S).
¬atT(T, L1, S) ← drive(T, L1, L2, S).
atT(T, L2, S) ← drive(T, L1, L2, S).

% precondition check
⊥ ← unload(P, T, L, S), not precondu(P, T, L, S).
precondu(P, T, L, S) ← atT(T, L, S−1),
                       in(P, T, S−1).
⊥ ← load(P, T, L, S), not precondl(P, T, L, S).
precondl(P, T, L, S) ← atT(T, L, S−1),
                       atP(P, L, S−1).
⊥ ← drive(T, L1, L2, S), not precondd(T, L1, L2, S).
precondd(T, L1, L2, S) ← atT(T, L1, S−1).
% inertia
atT(T, L, S) ← atT(T, L, S−1), not ¬atT(T, L, S).
atP(P, L, S) ← atP(P, L, S−1), not ¬atP(P, L, S).
in(P, T, S) ← in(P, T, S−1), not ¬in(P, T, S).
% goal check
⊥ ← goal(P, L), not atP(P, L, S), maxstep(S).
```

Fig. 26. Encoding for Package Delivery.

l_1, l_2) we cluster the packages into one abstract package, $\hat{p}_0$. Then, the abstract program will have the abstract actions $load(\hat{p}_0, t, l, s), unload(\hat{p}_0, t, l, s)$ which then lead to a plan

$$load(\hat{p}_0, t, l_1, 1), drive(t, l_1, l_2, 2), unload(\hat{p}_0, t, l_2, 3).$$

However, this plan is clearly spurious as no original action can match $load(\hat{p}_0, t, l_1, 1)$, which loads all packages in one step; thus many spurious answer sets will result. In order to avoid this, also abstraction over the *time* sort is necessary. By doing this, we can talk about abstract instances of actions and abstract from the concrete order of their application. Given that the sorts (i.e., blocks and time, respectively, packages and time) are independent, multiple calls of DASPAR to abstract over each sort one-by-one achieves the desired abstract program.

For the Package Delivery problem, consider two abstraction mappings $m_{package} = \{\{p_1, p_2, p_3, p_4\} \mapsto \hat{p}\}$ and $m_{time} = \{\{1, 2, 3, 4\} \mapsto \hat{t}_1, \{5\} \mapsto \hat{t}_2, \{6, 7, 8, 9\} \mapsto \hat{t}_3\}$. The constructed abstract program yields the abstract plan

$$load(\hat{p}, t, l_0, \hat{t}_1), drive(t, l_1, l_2, \hat{t}_2), unload(\hat{p}, t, l_2, \hat{t}_3)$$

which abstracts over the order of package (un)loading and includes abstract actions over time clusters.

Unfortunately, finding a suitable abstraction over multiple sorts, especially if one is over the *time* domain, is non-trivial. The abstraction over time via time clusters steers the plan computation and the action ordering. For example, for the time mapping $\{\{1\} \mapsto \hat{t}_1, \{2, 3\} \mapsto \hat{t}_2, \{4, 5, 6, 7, 8, 9\} \mapsto \hat{t}_3\}$ the abstract plan from above is spurious.

To summarize, while the use of abstraction in ASP planning appears to be attractive from a cognitive perspective, further research on several issues is needed in order to unleash the potential of this approach.

7.3. Related work

In the context of logic programming, abstraction has been considered many years back in the classic work of Cousot and Cousot [29]. However, the focus of their studies was on the use of abstract interpretations and termination analysis of programs, and moreover stable semantics was not addressed.

The work most related to our notion of abstraction in ASP are the simplification methods that strive for preserving the semantics. Such methods have been extensively studied over the years; we give here an overview of some notions. Notice that, different from these simplification methods, abstraction may lead to an over-approximation of the answer sets of a program, which changes the semantics, in a modified language.

Over-approximation by abstraction reduces the vocabulary which makes it different from *relaxation* methods [58,81]. These methods translate a ground program into its completion [24] and search for an answer set over the relaxed model. As they focus only on ground programs, they can be compared with the abstraction that omits atoms from the program, which does not need to account for loop formulas when searching for a concrete abstract answer set. However, finding the reason for spuriousness of an abstract answer set is trickier than finding the reason why a model of the program completion is not an answer set of the original program, since the abstract answer set contains fewer atoms and a search over the original program is needed to detect the reason why no matching answer set can be found.

7.3.1. Equivalence-based rewriting and program transformations in ASP

Equivalence of logic programs is considered under answer set semantics as follows: a program Π_1 is equivalent to a program Π_2 if $AS(\Pi_1) = AS(\Pi_2)$. *Strong equivalence* [80] is a much stricter condition: Π_1 and Π_2 are strongly equivalent

if, for any set R of rules, $\Pi_1 \cup R$ and $\Pi_2 \cup R$ are equivalent. This notion makes it possible to replace a part Q of a logic program by a strongly equivalent (simpler) program Q', without looking at the rest; [42,95,97,119] show ways of transforming programs by ensuring that the property holds. A more liberal notion is *uniform equivalence* [86,107] where R is restricted to a set of facts; that is Q and Q' are equivalent with respect to all factual inputs [40]. The notions of strong and uniform equivalence have been generalized to relativized equivalence [43], where the alphabet of R is restricted and further to a notion where in addition the occurrence of atoms in heads and bodies of the rules in R can be distinguished [122]; relativized strong equivalence with projections [56,104] allows to remove from answer sets auxiliary atoms. Relativized equivalence is related to our projected notion of concreteness, where certain (usually auxiliary) atoms are not considered for concreteness checking, see Section 3.3.4.

In terms of abstraction, there is the abstraction mapping that needs to be taken into account, since the constructed program may contain a modified language and the mapping may relate it back to the original language. Thus, to define equivalence between the original program Π and its abstraction Π^m according to a mapping m, we need to compare $m(AS(\Pi))$ with $AS(\Pi^m)$. The equivalence of Π and Π^m then becomes similar to the notion of faithfulness. However, as we have shown, even if the abstract program Π^m is faithful, refining m may lead to an abstract program having spurious answer sets. Thus, simply lifting the current notions of equivalence to abstraction may not achieve useful results.

Refinement-safe faithfulness, however, would allow one to use of Π^m instead of Π, as it preserves the answer sets. This property is achieved when the abstract program is unsatisfiable (which then implies that the original program was unsatisfiable). However, for original programs that are consistent, reaching an abstraction that is refinement-safe faithful is not easy; dividing the domain cluster may immediately cause a guess that introduces spurious solutions.

The notions of equivalence from above have been complemented with further ones that allow for relating logic programs over different alphabets. Correspondence frameworks for ASP programs [46] are triples of the form $\mathcal{F} = (\mathcal{U}, \mathcal{C}, \rho)$, where $\mathcal{U}$ is a set of propositional atoms, $\mathcal{C}$ (the context) is a class of programs over $\mathcal{U}$, and ρ is a binary relation over $2^{\mathcal{U}}$; two programs P and Q over $\mathcal{U}$ are then corresponding (with respect to $\mathcal{F}$) if for every $R \in \mathcal{C}$ the answer sets of $P \cup R$ and $Q \cup R$ are in relation ρ. These very generic frameworks allow one to capture the notions of equivalence from above; furthermore, the notion of over-approximation can be simply expressed by setting $\mathcal{C} = \{\emptyset\}$, i.e., to consist of the empty program, and $\rho_{\mathcal{U}'}$ to check for over-approximation of the answer sets of $P \cup R$ by the answer sets of $Q \cup R$, relative to a set of atoms $\mathcal{U}'$. However, while [46] discussed how to characterize equivalence for correspondence frameworks with $\rho_{\mathcal{U}'}$ in semantic terms using non-classical here-and-there models [98] and presented some general complexity results, the issue of how to obtain programs Q that over-approximate the answer sets of Q was not addressed; furthermore, the setting was propositional.

Synonymous theories [99,101] aimed to lift the notion of strong equivalence between programs to a setting where the programs have been formulated in different languages, but each language is bijectively interpretable in the other. To this end, the authors developed the notion of synonymous theories for quantified equilibrium logic, which is a well-known extension of the quantified logic of here-and-there that provides a logical reconstruction of answer set semantics [98, 100]. In this notion, the equilibrium models of synonymous theories T_1 and T_2 (representing logic programs) are in a one-to-correspondence and remain so under addition of new formulas in a suitable sense. It builds on definability and interpretation as in classical logic, which are extended to the non-classical setting. For Herbrand models as in our setting, synonymous theories do not allow for domain shrinking, and the faithful and bijective interpretation property preserves in a sense equilibrium models and does not permit strict over-approximation; the same would apply to non-Herbrand models with static domains, where elements would be clustered while preserving equivalence. Exploring our notion of over-approximation in the framework of [98,100] and to consider its possible application is an interesting issue for future research.

Other transformation methods, especially to help with grounding and solving of ASP programs, were investigated. A preprocessing technique was considered in [54] along with an assignment and a relation expressing equivalences among the parts of the program that could be assigned. Another form of preprocessing in [12,88] was applied to each rule of a program by computing a tree decomposition and then splitting the rule into multiple, smaller rules accordingly.

7.3.2. Abstraction in planning and agent verification

Starting from the early years of AI planning, applications of abstraction to help with the search and planning for complex domains have received a lot of attention. One main research focus has been on hierarchical planning, which considers different abstraction levels over the problem space. A plan is searched at the abstract level and then the solution is refined successively to more detailed levels in the abstraction hierarchy, until a concrete plan is computed at the original level. Sacerdoti [106] showed an abstraction notion that keeps the "critical" preconditions of actions and ignores the rest. For example, Knoblock [70] proposed an ordered monotonicity property to ensure that solving the subproblems by refining certain parts of the plan does not change the remainder of the abstract plan. A similar property was considered by Bacchus and Yang [7], which states that if the original problem is solvable, then any abstract solution must have a refinement. Anderson and Farley [4] constructed operator hierarchies by having classes of operators that share common effects and forming new abstract operators with the shared preconditions.

Another research focus has been on using abstractions to compute heuristics, which are estimates of the distances to the solution that guide the plan search. Pattern databases [31] are constructed from the results of projecting the state space to a set of variables of the planning task, called a pattern, which is to be solved optimally. The omission abstraction in [110] matches the intuition behind this projection notion. Edelkamp [38] was the first to apply this technique in planning.

He showed that a pre-compiled look-up table with the costs of abstract solutions can help the heuristic search in finding optimal solutions. The merge & shrink abstraction method of Helmert et al. [61] starts with a suite of single projections and then computes an abstraction by merging them and shrinking. A CEGAR-inspired method was proposed by Seipp and Helmert [116] based on cartesian abstractions, which form a general class of abstractions. The reason for the abstract plan being spurious is detected when trying to construct a concrete plan, and the abstraction is refined by splitting the states. Obtaining such a cartesian abstraction is also possible with domain abstraction introduced in Section 3; we further empower abstraction with a multi-dimensional handle in Section 5 that has the capability of representing a hierarchy of abstraction levels.

Giunchiglia and Walsh [59] presented a theory of abstractions which provided a basis to understand different types of abstractions, while characterizing abstractions as syntactic mappings between programs. Later, Nayak and Levy [90] considered a semantic theory, where first the original domain is abstracted and then the domain model gets abstracted to capture the abstracted domain. Their notion of model increasing (MI) abstractions is similar to our abstraction by over-approximation notion in ASP.

Although not investigated in detail, notions related to domain abstraction were also considered in heuristic-search planning. Hernádvölgyi and Holte [62] presented a domain abstraction notion over the states which are represented as fixed length vectors of labels; they also noted the possibility of encountering spurious states with some abstractions. Hoffman et al. [63] considered variable domain abstraction by modifying the add and delete lists of the operators accordingly. They argued that obtaining efficient results from abstraction in planning mostly relies on the how much irrelevance is in the problem; this is an observation we similarly made in our experiments. To further investigate the structure of problems that can obtain good results, especially in the context of ASP, is an interesting research direction.

The notion of irrelevant information and its effects were analyzed for planning by Nebel et al. [91], in which different heuristics were introduced to omit such information. Fox and Long [51] described a method for detecting symmetries in a problem which are then treated as indistinguishable to help the planner.

Abstraction was studied for situation calculus action theories by Banihashemi et al. [9], who imposed a bisimulation restriction on the abstraction in order to ensure that reasoning about the actions of an agent at the abstract level can be mapped to concrete reasoning. They later showed how this restricted notion of abstraction can be used in reasoning about a strategy for an agent to achieve a goal at the high level and then mapping it back into a low-level strategy [10]. However, their focus was not on how such an abstraction can be found.

For verifying the behavior of multi-agent systems, the use of abstraction has been investigated by Lomuscio et al. for abstracting over each agent to construct an abstract system while preserving the properties expressed in a temporal-epistemic logic [28] or alternating-time temporal logic [83]. In [28] the focus is not on how such an abstraction can be built. In [83], an abstraction over the states is made that have the same possible actions to execute and action abstraction keeps the actions of certain agents while omitting the rest. They considered a three-valued logic and the abstraction also preserves the behavior of not satisfying a property. Spuriousness may occur for the case of achieving an "uncertain" result for checking a specification in the abstract level, which then forces one to refine the abstraction by splitting the states after investigating the subformulas of the specification. They later extended this work to infinite state models [84] and abstracted them to finite models using predicate abstraction, and they presented an interpolant-based refinement method [11].

In the context of ASP and action languages, Dix et al. [35] proposed a way of formulating and solving hierarchical planning under the ASP semantics, with a focus on *ordered task decomposition*, which is planning each step in the order it will later be executed. For a particular application of mobile robot planning, Zhang et al. [123] performed hierarchical planning using the action language $\mathcal{BC}$.

7.3.3. Generalized planning

Finding a plan that can achieve the goal for a class of problem instances can give an understanding of the details relevant for these problems. The plan can then be used for any particular problem instance without the need for further search. Note that, as discussed in Section 7.2, the plans that are computed with our domain abstraction method can also be seen as generalized plans as they work for any original problem instance that maps to the abstract instance.

Srivastava et al. [118] proposed an abstraction method for constructing generalized plans with loops, by focusing on classical planning; however, selecting a good abstraction was beyond their scope. Bonet and Geffner [15] considered a setting where uncertainty is represented by a set of states, by clustering the states that provide the same observations. This view is similar to the indistinguishability notion we proposed in [109]. They studied the conditions for a policy (i.e., plan) to be general enough to work on other instances. Later they considered also trajectory constraints [14].

Illanes and McIlraith [65] studied abstraction for numeric planning problems by compiling them into classical planning. Recently, they used abstraction for problems with quantifiable objects [66], e.g., a number of packages to deliver to points A and B, to find by abstracting from the quantification generalized plans that work for multiple instances. For this, they built a quantified planning problem by clustering indistinguishable objects using reformulation techniques [105] to reduce symmetry, and then compute a general policy. While the quantifiability conditions of [66] restrict applicability, our method has the potential drawback of spurious answers.

8. Conclusion

Abstraction is an important aspect of Artificial Intelligence aiming at the omission of detail and fine-grained structure in problem solving, to reduce the cognitive and/or the computational complexity in order to better understand respectively effectively find solutions. In this spirit, we have introduced the notion of domain abstraction to Answer Set Programming, where the size of the domain of an ASP program shrinks while the collection of its answer sets is over-approximated, i.e., every original answer set can be mapped to some abstract answer set. We have shown how this can be applied to single or multiple sorts of the domain, and how multi-dimensionality can be handled that enables a hierarchical view of abstraction, with quad-tree abstractions as a showcase for multi-granular abstraction over grids.

More specifically, we have introduced two structure-preserving approaches that apply abstraction to the rules. The first approach keeps built-in relations in rules, which then must be lifted to the abstract domain, while the second approach, existential abstraction, loses their original format at the benefit of the ability to handle different levels of abstractions among the abstract elements, as needed in hierarchical abstraction.

As over-approximation may result in spurious abstract answer sets that do not correspond to original answer sets, we have presented a method for refining abstractions which uses ASP-debugging techniques to obtain hints for refinements and a CEGAR-style methodology of iterated abstraction refinement [25]. The approach has been implemented in tools DASPAR and mDASPAR (for multi-dimensional domain mappings), which given an ASP program and an initial abstraction, automatically refine it until for the induced abstract program either a concrete answer set is encountered or unsatisfiability is detected (which proves that the original program is unsatisfiable).

Our experiments showed the potential of the approach for understanding the core parts of an ASP program. In case of satisfiability, abstract answer sets focus on relevant details, as in case of planning problems such as Blocksworld and Package Delivery; a justification technique in ASP (cf. [20,103]) can be used to understand why a particular abstract answer set is computed, and moreover, if the abstraction is faithful, to identify details which are irrelevant for finding a solution. In case of unsatisfiability, the automatic abstraction refinement was able to catch the unsatisfiability without refining back to the original program. Furthermore, in grid-cell problems, a multi-dimensional view of abstraction enables zooming in to the area of the grid-cell which shows the reason for unsolvability; compared to the results of a small user study, explanations of decent quality were achieved, which suggests to continue this line of research.

8.1. Outlook

This article has provided seminal concepts and notions for domain abstraction in ASP, an assessment of semantic and computational properties, and results for a prototypical evaluation. The work on domain abstraction can be continued and extended in several directions.

One direction is to obtain more general notions of abstraction, and to apply abstraction to larger classes of ASP programs. As for the former, we remark that domain abstraction can be combined with omission abstraction [110] to obtain an abstraction that omits certain details and also abstracts over some part of the domain. This can be achieved with the current definitions, by first applying the desired domain abstraction to the program and then grounding the constructed non-ground abstract program to omit some of the atoms from it. The refinement decisions then need to take into account two causes for spuriousness; bad clustering of domain elements or bad omission of atoms. As regards larger classes of ASP programs, further language constructs like disjunction in rule heads, aggregates, or weak constraints as in the ASP Core-2 standard [21] are natural targets. Furthermore, extensions with nested rules, external atoms, or constraint solving are interesting other target languages.

An important aspect of the abstraction&refinement method is the inital abstraction mapping. Starting with too coarse abstractions may mislead the method into refining irrelevant parts of the abstraction. To overcome this, an understanding of a good initial abstraction needs to be investigated. Employing symmetry breaking techniques [34,37] in order to get hints on a good initial abstraction is a promising subject of future research. Furthermore, as the use of abstraction depends on the problem structure at hand, characterizations of different problem types and a better understanding of the effects of abstraction are necessary.

Another research direction concerns abstraction refinement. Different methods can be explored to help with the decision making in the refinement step. On the one hand, further heuristics for deciding about a refinement from a collection of abstract answer sets may be considered, where the range for local search may be increased and in addition domain-specific knowledge is exploited. On the other hand, by using justification methods such as [20,103] we can obtain an explanation of how an abstract answer set is built and check it on the original program. In case of failure, a reason for the spuriousness of the abstract justification may be distilled and exploited for abstraction refinement.

Related to this is the issue of abstraction assessment, where the question of what is a “good” abstraction needs to be further studied. Different criteria can be relevant in this respect, from technical ones like the degree of spuriousness (measured e.g. by the number of spurious answer sets) or the level of abstraction (measured e.g. by the granularity of clustering), to the cognitive appeal from a human user perspective in terms of understandability of the abstraction and the abstract program. Addressing the latter appears to be challenging and harder than developing measures for technical criteria, given that humans have implicit background knowledge about the domain.

A further research direction is advanced implementations beyond tuning the current prototypes. An apparent bottleneck is concreteness checking, which can be costly due to the need for grounding the original program in this process. Here one could explore the use of lazy-grounding, e.g. [22,72,96,121] and non-ground ASP solving, e.g. [6,87], or develop native techniques for concreteness checking. Further improvements may be using justification methods to explain abstract answer sets (as mentioned above), which as a further benefit mitigates the grounding issue and can yield substantial gains, or to embed the debugging program into the evaluation of the abstract program, such that hints can be obtained during the embedded checking; however, to achieve that this works efficiently is non-trivial.

Last but not least, the use of domain abstraction remains to be explored for applications. Different possibilities can be envisaged, with ASP program development as an obvious candidate. Different from common debugging techniques, domain abstraction aims to not just show the rules themselves that effect a certain behavior, but can moreover be used to identify the gist of the domain that is responsible for the latter and thus aids in gaining more insight into a program at hand. Another possible use of domain abstraction is as a solving technique to address scalability. While the state-of-the-art ASP solvers are quite efficient in solving problems, they may struggle with problems that create huge search spaces or require optimization. For such problems, abstraction could be useful. However, achieving a good abstraction that could help with solving is non-trivial, and advances in performance and in particular of concreteness checking would be necessary for a fruitful deployment. Finally, we believe that domain abstraction has potential for building systems that explain matters to a human end user, and thus can be a useful tool for realizing explainable AI. Our experiments with grid cell puzzles have nurtured this view, since the reasons for unsolvability obtained in an automated way by our tools are a good match with human intuition. However, this is just an initial step, and significant research efforts will have to be invested to make this view become reality.

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

We are grateful to the reviewers for their helpful and constructive comments to improve this work and its presentation.

Appendix A. Proofs

Proof of Theorem 3.1. Let $\hat{I}$ and $\widehat{\Pi}$ denote $m(I)$ and Π^m, respectively. Towards a contradiction, assume that there exists some $I \in AS(\Pi)$ s.t. $\hat{I} \cup \mathcal{T}_m \notin AS(\widehat{\Pi})$. This can occur either because (i) $\hat{I} \cup \mathcal{T}_m$ is not a model of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$ or (ii) $\hat{I} \cup \mathcal{T}_m$ is not a minimal model of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$.

(i) Suppose $\hat{I} \cup \mathcal{T}_m$ is not a model of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$. Then there exists some rule $\hat{r} \in \widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$ where $\hat{I} \cup \mathcal{T}_m \models B(\hat{r})$ and $\hat{I} \cup \mathcal{T}_m \not\models H(\hat{r})$. By construction of Π, $\hat{r}$ is only obtained by step (a), otherwise $\hat{r}$ would be a choice rule with head $H(\hat{r}) = \{m(\alpha)\}$, and $\hat{r}$ would be satisfied. Consequently $\hat{r}$ is a rule from step (a) for r in Π.

Since $\hat{I} \cup \mathcal{T}_m \models m(B^{std}(r)), \hat{d}_1 \circ \hat{d}_2, \tau_{\mathrm{I}}^{\circ}(\hat{d}_1, \hat{d}_2)$, we have $\hat{I} \cup \mathcal{T}_m \models m(B^{std}(r))$. If we have $p(\hat{e}_1, \ldots, \hat{e}_n) \in m(B^{std,+}(r))$, some $e_i \in \hat{e}_i$ exists such that $p(e_1, \ldots, e_n) \in I$ as all variables are standardized apart, $I \models B^{std,+}(r)$ for this choice. As for $p(\hat{e}_1, \ldots, \hat{e}_n) \in m(B^{std,-}(r))$, then $p(e_1, \ldots, e_n) \notin I$ for all $e_i \in \hat{e}_i$. So we can instantiate the abstract body $m(B^{std}(r))$ to some original body $B^{std}(r)$ where $I \models B^{std}(r)$. Also having $\hat{I} \models \hat{d}_1 \circ \hat{d}_2, \tau_{\mathrm{I}}^{\circ}(\hat{d}_1, \hat{d}_2)$ means $I \models d_1 \circ d_2$ for all $d_i \in \hat{d}_i$, thus we have $I \models B^{std}(r), d_1 \circ d_2$. So $r : \alpha \leftarrow B^{std}(r), d_1 \circ d_2$ is in Π^I. As I is a model, it follows that $I \models \alpha$, which then means $\hat{I} \models m(\alpha)$; this is a contradiction.

(ii) Suppose there exists some $\widehat{J} \subset \hat{I}$ such that $\widehat{J} \cup \mathcal{T}_m$ is a model of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$. We claim that $J = m^{-1}(\widehat{J}) \cap I$ is a model of Π^I; as $J \subset I$ holds, this would contradict that $I \in AS(\Pi)$. Assume $J \not\models \Pi^I$. Then J does not satisfy some rule $r : \alpha \leftarrow B^{std}(r), d_1 \circ d_2$ in Π^I, i.e., $J \models B^{std}(r), d_1 \circ d_2$ but $J \not\models \alpha$. As $J \subset I$ and I is a model of Π^I, we have $I \models \alpha$, thus, $\alpha \in I \setminus J$.

Now, we look at the cases for applying the mapping m to r, by considering the abstractions $m(B^{std}(r))$ and $m(d_1) \circ m(d_2)$, and show that a contradiction is always achieved.

First, assume that $\hat{I} \models m(B^{std}(r))$. There are the following cases for $m(J)$: (1-1) $m(J) \models m(B^{std}(r))$, or (1-2) $m(J) \not\models m(B^{std}(r))$.

(1-1) As $m(J) \models m(B^{std}(r))$, we look at $m(d_1) \circ m(d_2)$. We know that $J \models d_1 \circ d_2$.

- If $m(d_1) \circ m(d_2)$ has the relation type $\tau_{\mathrm{I}}^{\circ}(m(d_1), m(d_2))$, this means that we have $m(J) \models m(d_1) \circ m(d_2)$, and thus $m(J) \cup \mathcal{T}_m \models m(d_1) \circ m(d_1), \tau_{\mathrm{I}}^{\circ}(m(d_1), m(d_2))$. As $\widehat{J} = m(J)$ and $\widehat{J} \subset \hat{I}$, we also get $\hat{I} \cup \mathcal{T}_m \models m(d_1) \circ m(d_2), \tau_{\mathrm{I}}^{\circ}(m(d_1), m(d_2))$, thus the non-ground rule created by step (a) has an instantiation $m(\alpha) \leftarrow m(B^{std}(r)), m(d_1) \circ m(d_2), \tau_{\mathrm{I}}^{\circ}(m(d_1), m(d_2))$ in $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$. As $\widehat{J}$ and $\hat{I}$ are models of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$, we have $\widehat{J} \models m(\alpha)$ and $\hat{I} \models m(\alpha)$. Thus, $\alpha \in m^{-1}(\widehat{J})$ and $\alpha \in I$; by definition of J, we get $\alpha \in J$ thus $J \models \alpha$, which is a contradiction.
- If $m(d_1) \circ m(d_2)$ has the relation type $\tau_{\mathrm{III}}^{\circ}(m(d_1), m(d_2))$, this again means that we have $m(J) \models m(d_1) \circ m(d_2)$, and thus $m(J) \cup \mathcal{T}_m \models m(d_1) \circ m(d_2), \tau_{\mathrm{III}}^{\circ}(m(d_1), m(d_2))$ and $\hat{I} \cup \mathcal{T}_m \models m(d_1) \circ m(d_2), \tau_{\mathrm{III}}^{\circ}(m(d_1), m(d_2))$. Thus, as $m(\alpha) \in \hat{I}$

the non-ground choice rule created by step (b) amounts to $m(\alpha) \leftarrow m(B^{std}(r)), m(d_1) \circ m(d_2)$, $\tau^{\circ}_{\text{III}}(m(d_1), m(d_2))$ in $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$, which again achieves $\hat{J} \models m(\alpha)$, thus $J \models \alpha$, a contradiction.

- If $m(d_1) \circ m(d_2)$ has the relation type $\tau^{\circ}_{\text{IV}}(m(d_1), m(d_2))$, then we have $m(J) \not\models m(d_1) \circ m(d_2)$, i.e., $m(J) \models m(d_1) \,\overline{\circ}\, m(d_2)$, and thus $m(J) \cup \mathcal{T}_m \models m(d_1) \,\overline{\circ}\, m(d_2), \tau^{\circ}_{\text{IV}}(m(d_1), m(d_2))$. With similar reasoning as above, we find an instantiation of the non-ground rule created by step (c) and achieve $J \models \alpha$, a contradiction.

(1-2) We show that there is no possibility to have $\hat{J} \not\models m(B^{std}(r))$, for $\hat{J} = m(J)$, while $\hat{I} \models m(B^{std}(r))$. In order to have $\hat{J} \not\models m(B^{std}(r))$, some positive literal $\hat{\alpha_i} \in m(B^{std,+}(r))$ must occur in $\hat{I} \setminus \hat{J}$ so that $\hat{J} \not\models m(B^{std,+}(r))$. However, this contradicts with $J \models B^{std,+}(r)$.

Now, assume that $\hat{I} \not\models m(B^{std}(r))$. As $I \models B(r)$, we know that $\hat{I} \models m(B^{std,+}(r))$ holds. So we have the rule r in the form $l \leftarrow B^{std,+}(r), \mathit{not}\ \alpha_i, d_1 \circ d_2$ (according to restriction (R-1) on having at most one negative literal) where $\alpha_i \neq \alpha$ and $\hat{I} \not\models m(B^{std}(r))$ means $\hat{I} \models m(\alpha_i)$ for $\alpha_i \in B^{std,-}(r)$ while $I \not\models \alpha_i$, i.e., $\alpha_i \notin I$. So we get $\hat{I} \models m(B^{sh}_{\alpha_i}(r))$. Then there are the following cases for $m(J)$: (2-1) $m(J) \models m(B^{sh}_{\alpha_i}(r))$, or (2-2) $m(J) \not\models m(B^{sh}_{\alpha_i}(r))$.

(2-1) As we have $m(J) \models m(B^{sh}_{\alpha_i}(r))$, we look at $m(d_1) \circ m(d_2)$. We know that $J \models d_1 \circ d_2$.

- For cases $\tau^{\circ}_{\text{I}}(m(d_1), m(d_2))$ and $\tau^{\circ}_{\text{III}}(m(d_1), m(d_2))$, as we have $J \models d_1 \circ d_2$, we get $\hat{J} \models m(d_1) \circ m(d_2)$ and $\hat{I} \models m(d_1) \circ m(d_2)$. Notice that since $m(\alpha_i) \in \hat{I}$, there must be some $\alpha_i{}' \in I$ such that $m(\alpha_i) = m(\alpha_i{}')$, thus α_i is mapped to a non-singleton cluster $m(\alpha_i)$. So the atom $\mathit{not\ isSingleton}(m(j))$ holds true in $\hat{J}$ and $\hat{I}$ for some $j \in arg(\alpha_i)$ for which $|m^{-1}(m(j))| > 1$. Thus in $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$ we get an instantiation $m(\alpha) \leftarrow m(B^{sh}_{\alpha_i}(r)), m(d_1) \circ m(d_2), \mathit{not\ isSingleton}(m(j))$ of the non-ground rule created by (step d-i), and again achieve $J \models \alpha$, which is a contradiction.
- For the case $\tau^{\circ}_{\text{IV}}(m(d_1), m(d_2))$, with similar reasoning as in (1-1), we find instantiations of the non-ground rules created by (step d-ii) and achieve $J \models \alpha$, which is a contradiction.

(2-2) We show that there is no possibility to have $m(J) \not\models m(B^{sh}_{\alpha_i}(r))$, while $\hat{I} \models m(B^{sh}_{\alpha_i}(r))$. As $J \models B(r)$, we know that $m(J) \models m(B^{std,+}(r))$ holds. So $m(J) \not\models m(B^{sh}_{\alpha_i}(r))$ means $m(J) \not\models m(\alpha_i)$ while $\hat{I} \models m(\alpha_i)$. Now, we take a look at Π^I. As there must be some $\alpha_i{}' \in I$ (such that $m(\alpha_i) = m(\alpha_i{}')$), this means that there is some rule $r' : \alpha_i{}' \leftarrow B^{std}(r'), d'_1 \circ d'_2$ in Π^I. We then take a look at the abstraction of r'. By doing the same case analysis of (1-1), (1-2) and (2-1), we achieve $m(J) \models m(\alpha_i{}')$, i.e., $m(J) \models m(\alpha_i)$, which yields a contradiction. As for (2-2), this means the rule r' is of form $r' : \alpha_i{}' \leftarrow B^{std,+}(r'), \mathit{not}\ \alpha_{i_2}, d'_1 \circ d'_2$ where we want to claim $m(J) \not\models m(\alpha_{i_2})$. For this, we take a look at another rule r'' in Π^I of form $r'' : \alpha'_{i_2} \leftarrow B^{std}(r''), d''_1 \circ d''_2$ with $m(\alpha'_{i_2}) = m(\alpha_{i_2})$. By restriction (R-3) on no negative cyclic dependency among the atoms, this recursive process eventually ends, say, after n steps, at some rule $r'^n : \alpha'_{i_n} \leftarrow B^{std}(r'^n), d'^n_1 \circ d'^n_2$ where case (2-2) is not applicable, and $m(J) \models m(\alpha'_{i_n})$ is achieved. Then by tracing the rules back to r we get $m(J) \models m(\alpha_i)$. Thus $m(J) \not\models m(B^{sh}_{\alpha_i}(r))$ is not possible. □

Proof of Theorem 3.2. Similar to proof of Theorem 3.1, we assume towards a contradiction that there exists some $I \in AS(\Pi)$ such that $\hat{I} \cup \mathcal{T}_m \notin AS(\widehat{\Pi})$. This can occur either because (i) $\hat{I} \cup \mathcal{T}_m$ is not a model of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$ or (ii) $\hat{I} \cup \mathcal{T}_m$ is not a minimal model.

(i) Let $\hat{I} \cup \mathcal{T}_m$ be not a model of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$. Then there exists some rule $\hat{r} \in \widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$ where $\hat{I} \cup \mathcal{T}_m \models B(\hat{r})$ and $\hat{I} \cup \mathcal{T}_m \not\models H(\hat{r})$. For cases (G-1) and (G-3), the contradiction is achieved similar to the proof of Theorem 3.1, since $\hat{r}$ is a rule from step (a). As for case (G-2), we will have $\hat{I} \cup \mathcal{T}_m \models rel'(\hat{d}), \tau^{rel'}_{\text{I}}(\hat{d})$, where $\hat{d}$ is a shorthand for $\hat{d}_{1,1}, \hat{d}_{2,1}, \ldots, \hat{d}_{1,k}, \hat{d}_{2,k}$; then by definition of rel' this means $I \models B^{std}(r), d_{1,1} \circ_1 d_{2,1}, \ldots, d_{1,k} \circ_k d_{2,k}$ for some $d_{1,1} \in \hat{d}_{1,1}, d_{2,1} \in \hat{d}_{2,1}, \ldots, d_{1,k} \in \hat{d}_{1,k}, d_{2,k} \in \hat{d}_{2,k}$ and r in Π^I. This reaches a contradiction as I is a model and $I \models \alpha$, which means $\hat{I} \models m(\alpha)$.

(ii) Now let there be $\widehat{J} \subset \hat{I}$ such that $\widehat{J} \cup \mathcal{T}_m$ is a model of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$. We claim that $J = m^{-1}(\widehat{J}) \cap I \subset I$ is a model of Π^I; which would contradict that $I \in AS(\Pi)$. Assume $J \not\models \Pi^I$. J does not satisfy some rule $r : \alpha \leftarrow B^{std}(r), rel'(d)$ in Π^I, i.e., $J \models B^{std}(r), d_{1,1} \circ_1 d_{2,1}, \ldots, d_{1,k} \circ_k d_{2,k}$ but $J \not\models \alpha$, i.e., $\alpha \notin J$. As $J \subset I$ and I is a model of Π^I, we have $I \models \alpha$, i.e., $\alpha \in I \setminus J$. We consider the abstractions $m(B^{std}(r))$ and $m(d_{1,1}) \circ_1 m(d_{2,1}), \ldots, m(d_{1,k}) \circ_k m(d_{2,k})$.

First, assume $\hat{I} \models m(B^{std}(r))$. There are the following cases for $m(J)$: (1-1) $m(J) \models m(B^{std}(r))$, or $m(J) \not\models m(B^{std}(r))$.

(1-1) As $m(J) \models m(B^{std}(r))$, we look at $m(d_{1,1}) \circ_1 m(d_{2,1}), \ldots, m(d_{1,k}) \circ_k m(d_{2,k})$. We know that $J \models d_{1,1} \circ_1 d_{2,1}, \ldots, d_{1,k} \circ_k d_{2,k}$.

(1-1-a) If all $m(d_{1,i}) \circ_i m(d_{2,i})$ have the relation type $\tau^{\circ_i}_{\text{I}}(m(d_{1,i}) \circ_i m(d_{2,i}))$, this means that we have $m(J) \models m(d_{1,1}) \circ_1 m(d_{2,1}), \ldots, m(d_{1,k}) \circ_k m(d_{2,k})$, and thus

$$m(J) \cup \mathcal{T}_m \models rel'(\hat{d}), \tau^{rel'}_{\text{I}}(\hat{d}). \tag{93}$$

As $\hat{J} = m(J)$ and $\widehat{J} \subset \hat{I}$, we also get $\hat{I} \cup \mathcal{T}_m \models rel'(\hat{d}), \tau^{rel'}_{\text{I}}(\hat{d})$, thus the non-ground rule created by step (a) has an instantiation $m(\alpha) \leftarrow m(B^{std}(r)), rel'(\hat{d}), \tau^{rel'}_{\text{I}}(\hat{d})$ in $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$. As $\widehat{J}$ and $\hat{I}$ are models of $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$, we have $\widehat{J} \models m(\alpha)$ and $\hat{I} \models m(\alpha)$. Thus, by definition of J, we get $J \models \alpha$, which is a contradiction.

(1-1-b) If at least one $m(d_{1,i}) \circ_i m(d_{2,i})$ has the relation type $\tau^{\circ_i}_{\text{III}}(m(d_{1,i}), m(d_{2,i}))$, while no $m(d_{1,j}) \circ_j m(d_{2,j})$ has the relation type $\tau^{\circ}_{\text{IV}}(\hat{d}_{1,j}, \hat{d}_{2,j})$, above (93) is achieved for $\tau^{rel'}_{\text{III}}(\hat{d})$ in place of $\tau^{rel'}_{\text{I}}(\hat{d})$.

(1-1-c) If at least one $m(d_{1,i}) \circ_i m(d_{2,i})$ has the relation type $\tau^{\circ_i}_{\text{IV}}(m(d_{1,i}), m(d_{2,i}))$, this means that we have $m(J) \models \ldots, m(d_{1,i}) \circ_i m(d_{2,i}), \ldots$. Thus, for $rel'(\hat{d})$ we have $m(J) \not\models rel'(\hat{d})$ but $m(J) \models \overline{rel}'(\hat{d})$, and

$$m(J) \cup \mathcal{T}_m \models \overline{rel}'(\hat{d}), \tau^{rel'}_{\text{IV}}(\hat{d}). \tag{94}$$

By the same reasoning in (1-1-a), we get that the non-ground choice rule created by step (c) amounts to $m(\alpha) \leftarrow m(B^{std}(r)), \overline{rel}'(\hat{d}), \tau^{rel'}_{\text{IV}}(\hat{d})$ in $\widehat{\Pi}^{\hat{I} \cup \mathcal{T}_m}$, and thus we reach a contradiction.

(1-2) This case is handled the same as in proof (1-2) of Theorem 3.1.

Now, we focus on the case $\hat{I} \not\models m(B^{std}(r))$. As $I \models B(r)$, we know that $\hat{I} \models m(B^{std,+}(r))$ should hold. Then $\hat{I} \not\models m(B^{std}(r))$ means that for a non-empty set $L \subseteq B^{std,-}(r)$ of atoms, we have $\hat{I} \models \alpha_i$ for each $\alpha_i \in L$, $1 \leq i \leq n$, while $I \not\models \alpha_i$, i.e., $\alpha_i \notin I$. Assume we further have a set L_{c_j} of atoms involved in a negative cycle with α. Since in $B^{sh}_{L,L_{c_j}}(r)$ the literals α_i in L either get shifted or get omitted if $\{\alpha_i, \alpha\} \in L_{c_j}$ (by (42)), we get $\hat{I} \models m(B^{sh}_{L,L_{c_j}}(r))$ and $\hat{I} \models m(not\ B^{std,-}(r) \setminus L)$.

Then there are the following cases for $m(J)$: (2-1) $m(J) \models m(B^{sh}_{L,L_{c_j}}(r))$ and $m(J) \models m(not\ B^{std,-}(r) \setminus L)$, or (2-2) $m(J) \not\models m(B^{sh}_{L,L_{c_j}}(r))$ or $m(J) \not\models m(not\ B^{std,-}(r) \setminus L)$.

(2-1) As $m(J) \models m(B^{sh}_{L,L_{c_j}}(r))$ and $m(J) \models m(not\ B^{std,-}(r) \setminus L)$, similar to proof (1-1) above and (2-1) of Theorem 3.1, the abstraction $\hat{d}_{1,1} \circ_1 \hat{d}_{2,1}, \ldots, \hat{d}_{1,k} \circ_k \hat{d}_{2,k}$ on relations is considered, and the contradiction $J \models l$ is achieved.

(2-1-a) By (1-1-a) and (1-1-b), we get the case (93) and same for $\tau^{rel'}_{\text{III}}(\hat{d})$. We know that for each $\alpha_i \in L$, $1 \leq i \leq n$ as $m(J) \models m(\alpha_i)$ and as $\alpha_i \notin J$ (since $J \subseteq I$ and $\alpha_i \notin I$), this means that *not isSingleton*$(m(j_i))$ holds true in $m(J)$ and $\hat{I}$ for some $j_i \in arg(\alpha_i)$. Thus we have $m(J) \cup \mathcal{T}_m \models rel'(\hat{d})$, *not isSingleton*$(m(j_1))$, …, *not isSingleton*$(m(j_n))$, which means that $m(\alpha) \in \hat{I}$, thus $\alpha \in I$ and by definition of J, $\alpha \in J$, which is a contradiction.

(2-1-b) By (1-1-c), we get the case (94) and by a similar reasoning as in (2-1-a) we also have $m(J) \models$ *not isSingleton*$(m(j_1)), \ldots,$ *not isSingleton*$(m(j_n))$, hence $m(J) \cup \mathcal{T}_m \models \overline{rel}'(\hat{d}), \tau^{rel'}_{\text{IV}}(\hat{d})$, *not isSingleton*$(\hat{j}_1)$, …, *not isSingleton*$(\hat{j}_n)$. Thus we similarly achieve a contradiction.

(2-2) We first show that there is no possibility to have $m(J) \not\models m(B^{sh}_{L,L_{c_j}}(r))$, while $\hat{I} \models m(B^{sh}_{L,L_{c_j}}(r))$. As $J \models B(r)$, we know that $m(J) \models m(B^{std,+}(r))$ should hold. So $m(J) \not\models m(B^{sh}_{L,L_{c_j}}(r))$ means $m(J) \not\models m(\alpha_i)$ for some $\alpha_i \in L \setminus L_{c_j}$, showing that $\{\alpha_i, \alpha\} \notin L_{c_j}$, while $\hat{I} \models m(\alpha_i)$. Now, we take a look at Π^I. As there must be some $\alpha_i{}' \in I$ (such that $m(\alpha_i) = m(\alpha_i{}')$), this means that there is some rule $r' : \alpha_i{}' \leftarrow B^{std}(r'), d'_{1,1} \circ_1 d'_{2,1}, \ldots, d'_{1,k} \circ_k d'_{2,k}$ in Π^I. We then take a look at the abstraction of r'. By doing the same case analysis of (1-1), (1-2) and (2-1), we achieve $m(J) \models m(\alpha_i{}')$, i.e., $m(J) \models m(\alpha_i)$, which yields a contradiction. As for (2-2) (i.e., $m(J) \not\models m(B^{sh}_{L,L_{c_{j'}}}(r'))$ for some $L \subseteq B^{std,-}(r)$ and for some $L_{c_{j'}}$ with $\alpha_i{}' \in L_{c_{j'}}$), this means the rule r' is of form

$$r' : \alpha_i{}' \leftarrow B^{std,+}(r'), B^{std,-}(r'), d'_{1,1} \circ_1 d'_{2,1}, \ldots, d'_{1,k} \circ_k d'_{2,k},$$

where some $l_{i_2} \in B^{std,-}(r)$ exists s.t. $l_{i_2} \in L \setminus L_{c_{j'}}$ (i.e., $\{l_{i_2}, \alpha_i{}'\} \notin L_{c_{j'}}$) and we want to claim $m(J) \not\models m(l_{i_2})$. For this, we take a look at another rule r'' in Π^I of form $r'' : \alpha'_{i_2} \leftarrow B^{std}(r''), d''_{1,1} \circ_1 d''_{2,1}, \ldots, d''_{1,k} \circ_k d''_{2,k}$ with $m(\alpha'_{i_2}) = m(\alpha_{i_2})$. Since this backwards traverse is done over the negated literals not involved in negative cycles with the head of the respective rules, this recursive process eventually ends, say, after n steps, at some rule $r'^n : \alpha'_{i_n} \leftarrow B^{std}(r'^n), d'^n_{1,1} \circ_1 d'^n_{2,1}, \ldots, d'^n_{1,k} \circ_k d'^n_{2,k}$ where case (2-2) is not applicable, and $m(J) \models m(\alpha'_{i_n})$ is achieved. Then by tracing the rules back to r we get that $m(J) \models m(\alpha_i)$ actually holds. Thus $m(J) \not\models m(B^{sh}_{L,L_{c_j}}(r))$ is not possible.

The case $m(J) \not\models m(not\ B^{std,-}(r) \setminus L)$ also cannot hold, since that would mean that there exists some $\alpha_i \in B^{std,-}(r) \setminus L$ such that $m(J) \models m(\alpha_i)$, while $\hat{I} \not\models m(\alpha_i)$ (since $\hat{I} \models m(not\ B^{std,-}(r) \setminus L)$), which is a contradiction to $m(J) \subset \hat{I}$. □

Proof of Lemma 3.9. By definition, we need to check (1) that $\hat{I}$ is a model of $(\Pi^m)^{\hat{I}}$ and (2) that $\hat{I}$ is minimal, no $\hat{J} \subset \hat{I}$ is a model of $(\Pi^m)^{\hat{I}}$.

As for (1), we can refute the property by guessing a rule $r \in \Pi^m$ and a variable substitution θ and verifying that $\hat{I}$ does not satisfy $(r\theta)^{\hat{I}}$, where $r\theta$ denotes the ground instance of r obtained by applying θ to its variables; note that in this case $(r\theta)^{\hat{I}} \in (\Pi^m)^{\hat{I}}$ holds.

Each rule $r \in \Pi^m$ has polynomial size in the input. Checking whether $r \in \Pi^m$ holds is feasible in polynomial time, as computing the independent negative cycles $L_{c_1}, \ldots, L_{c_l}$ of the program Π is feasible in polynomial time as well. Furthermore,

checking whether an instance $r' = r\theta$ of r over the abstract domain is in the GL-reduct $(\Pi^m)^{\hat{I}}$ of the abstract program Π^m with respect to $\hat{I}$ is feasible in polynomial time. Overall, refuting (1) is consequently in NP.

As for (2), $\hat{I}$ is minimal if each atom $a \in I$ has a proof, given by a sequence $r_1, r_2, \ldots r_k$ of applications of rules from $r_i \in (\Pi^m)^{\hat{I}}$ where each positive body literal of r_i occurs in some head of r_j, $j < i$. Note that w.l.o.g. $\hat{I} = \{a_1, \ldots, a_k\}$ and a_i has as proof $r_1, \ldots, r_i$, $i = 1, \ldots, k$. As the proof can be guessed and nondeterministically verified in polynomial time, it follows that (2) is in NP. Hence it follows that the problem is in Δ_2^p (more precisely, in the class DP). □

Proof of Theorem 3.10. To show that $\hat{I}$ is a concrete abstract answer set of Π^m, we can guess an interpretation J of Π and check that (a) $m(J) = \hat{I}$, (b) $m(J) \in AS(\Pi^m)$, and (c) $J \in AS(\Pi)$. Testing (a) is clearly polynomial in the size of J, and by Lemma 3.9, (b) and (c) are feasible in Δ_2^p in the size of J and Π (and thus in exponential time in the size of $\hat{I}$ and Π); consequently, deciding whether $\hat{I}$ is a concrete abstract answer set of Π^m is in NEXP. For bounded predicate arities, the guess for J has polynomial size in the input, and we can check the conditions (b) and (c) by Lemma 3.9 with an NP oracle in polynomial time; this establishes Σ_2^p membership.

The matching lower bounds are shown by a reduction from deciding whether a given non-ground program Π has some answer set, which is NEXP-complete in the general case and Σ_2^p-complete for bounded predicate arities [33,39].

Without loss of generality, Π involves a single predicate p (which can be achieved by reification and padding arguments) and contains some fact $p(\vec{d})$. The mapping we define is $m = \{\{d_1, \ldots, d_n\} \mapsto \hat{d}\}$ where $d_1, \ldots, d_n$ form the Herbrand domain. Then $\hat{I} = \{p(\hat{d}, \ldots, \hat{d})\}$ is a concrete abstract answer set of Π^m iff Π has some answer set. Note that actually $\hat{I} \in AS(\Pi^m)$ holds; thus the overall complexity does not change if this property is asserted. This proves the result. □

Proof of Theorem 3.12. For the membership, one can guess an interpretation $\hat{I}$ of Π^m such that $\hat{I}$ is an answer set of Π^m, and then check whether $\hat{I}$ is spurious. By Theorem 3.10, the spuriousness check can be done with a coNEXP oracle in general and with a Σ_2^p oracle in the bounded predicate case. However, by applying standard padding techniques,[10] it follows that a coNP oracle is sufficient in the general case. This proves membership of the problem in NEXP^{NP} in the general case and in Σ_3^p in the bounded predicate case, respectively.

The NEXP^{NP}-hardness in the general case is shown by a reduction from evaluating second-order logic formulas of a suitable form over finite relational successor structures, i.e., relational structures $S = (D, R^S)$ with a universe D and interpretations R_i^S for the relations R_i in $R = R_1, \ldots, R_k$, which include the relations *first*(x), *next*(x, y) and *last*(x) associated with a linear ordering $\leq$ of D.

Lemma A.1. *Given a second-order (SO) sentence of the form $\Phi = \exists P \forall Q\, .\varphi$ where $P = P_1, \ldots, P_{m_1}$ and $Q = Q_1, \ldots, Q_{m_2}$ are predicate variables and $\varphi = \bigvee_j \varphi_j$ is FO such that each φ_j is of the form $\varphi_j = \exists x_1, \ldots, x_n l_{j,1} \wedge \cdots \wedge l_{j,k}$ where each $l_{i,j}$ is a FO-literal, and a finite relational successor structure S, deciding whether $S \models \Phi$ is NEXP^{NP}-complete.*

This lemma can be obtained from the facts that (1) evaluating SO-sentences of the form $\Psi = \exists P \forall Q\, .\varphi$, where φ is a first-order formula, over finite relational successor structures is NEXP^{NP}-complete, cf. [60], and (2) that Ψ can be transformed into some Φ of the form described in polynomial time; the latter is possible using second-order skolemization and auxiliary predicates for quantifier elimination, cf. [44] and for denoting subformulas, such that $\varphi(\vec{x}) \equiv P_\varphi(\vec{x})$ and $\varphi(\vec{x}) = \varphi_1(\vec{x}) \wedge \varphi_2(\vec{x})$ is expressed by $P_\varphi(\vec{x}) \equiv P_{\varphi_1}(\vec{x}) \wedge P_{\varphi_2}(\vec{x})$ etc.

We first describe how to encode evaluating the sentence $\Phi' = \exists P \exists Q \neg\varphi$ into an ordinary program Π_0, and then extend the encoding to prove the result. We define the rules of Π_0 as follows, where D serves as a domain predicate for constants $\mathcal{C} = \{x_1, \ldots, x_n\}$:

$$P_{j,i}(X_1, \ldots, X_n) \leftarrow \mathit{not}\ \overline{P_{j,i}}(X_1, \ldots, X_n), D(X_1), \ldots, D(X_n). \quad \text{for each } P_{j,i} \in P \tag{95}$$

$$\overline{P_{j,i}}(X_1, \ldots, X_n) \leftarrow \mathit{not}\ P_{j,i}(X_1, \ldots, X_n), D(X_1), \ldots, D(X_n). \quad \text{for each } P_{j,i} \in P \tag{96}$$

$$Q_{j,i}(Y_1, \ldots, Y_n) \leftarrow \mathit{not}\ \overline{Q_{j,i}}(Y_1, \ldots, Y_n), D(Y_1), \ldots, D(Y_n). \quad \text{for each } Q_{j,i} \in Q \tag{97}$$

$$\overline{Q_{j,i}}(Y_1, \ldots, Y_n)\} \leftarrow \mathit{not}\ Q_{j,i}(Y_1, \ldots, Y_n), D(Y_1), \ldots, D(Y_n). \quad \text{for each } Q_{j,i} \in Q \tag{98}$$

$$\mathit{sat} \leftarrow l_{j,1}^{\neg/\mathit{not}} \wedge \cdots \wedge l_{j,k}^{\neg/\mathit{not}}. \tag{99}$$

$$\mathit{ok} \leftarrow \mathit{not}\ \mathit{ok}. \tag{100}$$

$$\mathit{ok} \leftarrow \mathit{not}\ \mathit{sat}. \tag{101}$$

where $l^{\neg/\mathit{not}}$ denotes the replacement of $\neg$ in l by *not*.[11]

[10] The input x to the oracle is changed to (x, y), where y is an (exponentially) long string y, and the oracle query considers x from the input only. This artificially lowers the time bound within the query (measured in the size of (x, y)) can be answered.

[11] To make the rules (99) safe, in their body atoms $D(X)$ can be added for unsafe variables X.

Informally, the rules (95), (96) and (97), (98) guess extensions for the predicates in P and Q, respectively, while the rules (99) evaluate the formula φ. A guess for P and Q yields an answer set of Π_0 augmented with S (provided as positive facts) iff φ evaluates over S to false; in this case, no rule (99) fires and thus *sat* cannot be derived, which means in turn that *ok* can be derived by (101) and thus the constraint (100) is satisfied. On the other hand, deriving *ok* is necessary to have an answer set, which means that *sat* must not be derived from the guess for P and Q.

We extend the program Π_0 now for spuriousness checking. To this end, we introduce for the domain $D = \{x_1, \ldots, x_n\}$ at hand a copy $D' = \{y_1, \ldots, y_n\}$ and link x_i to y_i via a predicate $eq(x, y)$ that holds for $x, y \in D \cup D'$ iff $x = x_i \wedge y = y_i$ for some $i = 1, \ldots, n$. The idea is to use D and D' in the predicates from P and Q, respectively, and to abstract D' into a single element, such that for every guess χ for P, some abstract answer set $\hat{I}_\chi$ of the abstract program Π^m will exist; and that, moreover, $\hat{I}_\chi$ will be concrete if for some guess for Q, we have an answer set of Π, where the latter program is equivalent to Π_0; thus $\hat{I}_\chi$ will be spurious iff no guess for Q will yield an answer set of Π_0, which means that the formula $\exists P \forall Q . \varphi$ evaluates to true.

We make the following adjustments.

1. First, we replace in (97) and (98) the predicate D with D'.
2. Next, for each rule r from (99) we add for each term t that occurs in the rule body a "typing" atom $D(t)$, we replace each term t that occurs in a Q-literal with a fresh variable X_t and add the atoms $D'(X_t)$ and $eq(t, X_t)$.
3. To each rule r obtained from the previous step we add *not* $succ(y_1, y_1)$ in the body; this literal evaluates to true with no abstraction, while under abstraction it will cause uncertainty and thus lead to a choice rule.
4. We add facts $eq(x_i, y_i)$ and $D(x_i)$, $D'(y_i)$, for $i = 1, \ldots, n$.
5. We add facts $Q_{j,i}(y_0, \ldots, y_0)$, $\overline{Q_{j,i}}(y_0, \ldots, y_0)$ for all $Q_{j,i} \in Q$, where y_0 is a fresh constant.

It is not hard to establish that the answer sets I of the resulting program Π (over S) correspond to the answer sets I_0 of Π_0 over S; each I is obtained from some I_0 by replacing in the $Q_{j,i}$- and $\overline{Q_{j,i}}$-atoms the constant x_l with the corresponding y_l, adding all facts $Q_{j,i}(y_0, \ldots, y_0)$, $\overline{Q_{j,i}}(y_0, \ldots, y_0)$, and adding the $eq(x_i, y_i)$, $D'(y_j)$ facts, $i = 1, \ldots, n$.

The mapping that we construct is $m = \{\{x_1\} \mapsto x_1, \ldots, \{x_n\} \mapsto x_n\} \cup \{\{y_0, y_1, \ldots, y_n\} \mapsto \hat{y}\}$. In the abstract program Π^m, the rules (95), (96) are carried over, while the modified rules (97), (98) are turned into rules to derive abstract atoms over $Q_{j,i}$ resp. $\overline{Q_{j,i}}$. However, since Π^m contains the abstracted facts $Q_{j,i}(\hat{y}, \ldots, \hat{y})$, $\overline{Q_{j,i}}(\hat{y}, \ldots, \hat{y})$, these rules are redundant.

The modified rules (99) are turned into guessing rules for *sat*, while the other rules (100) and (101) remain unchanged. The abstract answer sets of Π^m correspond to guesses χ for P to which *ok* and all $Q_{j,i}(\hat{y}, \ldots, \hat{y})$, $\overline{Q_{j,i}}(\hat{y}, \ldots, \hat{y})$ are added (*sat* is guessed false); denote this answer set by I_χ.

The answer set I_χ is concrete, if there is some guess μ for Q such that we obtain an answer set I of the program Π that is mapped to I_χ, i.e., $m(I_1) = I_\chi$; this I_1 corresponds to some answer set I_0 as described above. Thus I_χ is spurious, if no such guess μ for Q exists.

Putting it all together, it holds that Π has with respect to the mapping $m = \{\{x_1\} \mapsto x_1, \ldots, \{x_n\} \mapsto x_n\} \cup \{\{y_0, y_1, \ldots, y_n\} \mapsto \hat{y}\}$ some spurious answer iff the formula Φ in Lemma A.1 evaluates over S to true. Since Π and m are constructable in polynomial time from Φ and S, this proves coNEXP$^{\text{NP}}$ hardness in the general case.

For the bounded predicate arities case, the evaluation of a formula Φ as in Lemma A.1 is Σ_3^p-complete; furthermore, all steps in producing the program Π preserves bounded arities. Thus with the same argument, we obtain Σ_3^p-hardness for deciding whether some spurious answer set exists for bounded predicate arities. This proves the result. □

Proof of Proposition 4.2. Let X be any interpretation over $\mathcal{V}$ such that $m(X) = \hat{I}$. We will show that with the help of the auxiliary rules/atoms, some interpretation X' over $\mathcal{V}_{debug}$ which is a minimal model of $\Pi'^{X'}$ where $\Pi' = \Pi_{debug} \cup Q_{\hat{I}}^m$ can be built starting from X. To this end, we first add all facts α from Π to X and all atoms ko_{n_r}, where $H(r) = \bot$ or $B(r) \neq \emptyset$ for a rule in $r \in grd(\Pi)$. The resulting X' will satisfy all rules in $grd(\Pi_{debug})$, and thus in $\Pi_{debug}^{X'}$, that are facts from Π or have either *not* ko_{n_r} in the body or ko_{n_r} in the head. Furthermore, by construction X' will satisfy $Q_{\hat{I}}^m$. We now satisfy the remaining rules (i) in $\mathcal{T}_{meta}[\Pi]$ by adding ap_{n_r} and bl_{n_r} atoms, in this order and (ii) in $\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{V}]$ by auxiliary atoms $\overline{p}(c_1, \ldots, c_n)$ for the choice-rules and ab_* atoms for the deactivation/activation rules.

(i) For every instance r' of a rule $r \in \Pi$ such that $X' \models B(r')$, we add to X' (a) the atom $ap_{n_r}(c_1, \ldots, c_n)$ where $arg(H(r')) = \{c_1, \ldots, c_n\}$, if $H(r) \neq \bot$ and $n > 0$, and (b) the atoms ap_{n_r}, $ap_{n_r}(d_1, \ldots, d_{n'})$ where $arg(B(r')) = \{d_1, \ldots, d_{n'}\}$ if $H(r) = \bot$ or $B(r) = 0$. After that, we add to X' all ground atoms $bl_{n_r}(c_1, \ldots, c_n)$ and bl_{n_r} such that $ap_{n_r}(c_1, \ldots, c_n) \notin X'$ and $ap_{n_r} \notin X'$, respectively.

(ii) For every instance r' of a rule $r \in \Pi$ with $B(r) \neq \emptyset$ and $H(r) \neq \bot$ where $arg(H(r')) = \{c_1, \ldots, c_n\}$, if $ap_{n_r}(c_1, \ldots, c_n) \in X'$ and $H(r') \notin X'$ we add $\overline{H(r')}$ and $ab\_deact_{n_r}(c_1, \ldots, c_n)$ to X'; furthermore, for every instance r' of a rule $r \in \Pi$ with $H(r) = \bot$ where $arg(B(r')) = \{d_1, \ldots, d_{n'}\}$, if $ap_{n_r}(d_1, \ldots, d_{n'}) \in X'$ we add $ab\_deactCons_{n_r}(d_1, \ldots, d_{n'})$ to X'. Finally, for every atom $p(c_1, \ldots, c_n) \in \mathcal{A}$, if no instance r' of a rule $r \in \Pi$ exists such that $H(r') = p(c_1, \ldots, c_n)$ and $X' \models B(r')$ (that is, $bl_{n_r}(c_1, \ldots, c_n) \in X'$ for all rules r defining p), then we add (a) $\overline{p}(c_1, \ldots, c_n)$ to X' if $p(c_1, \ldots, c_n) \notin X'$ and (b) $ab\_act_p(c_1, \ldots, c_n)$ to X' if $p(c_1, \ldots, c_n) \in X'$.

As X' and X coincide on $\mathcal{V}$ (i.e., $X' \cap HB_\Pi = X \cap HB_\Pi = X$) we have $X' \models Q_{\hat{I}}^m$ as claimed, and thus $X' \models \Pi'^{X'}$. To show that X' is a minimal model of $\Pi'^{X'}$, we can by [75] show that Π' has no unfounded set U w.r.t. X' such that $U' := U \cap X' \neq \emptyset$, i.e., no $U \subseteq HB_{\Pi'}$ such that $U' \neq \emptyset$ and for each atom $\alpha \in HB_{\Pi'}$ and each $r' \in grd(\Pi')$, either $X' \not\models B(r')$ or $B^+(r') \cap U \neq \emptyset$.

Towards a contradiction, suppose such an U exists. By construction of X', it is easily checked that for every $\alpha \in X'$, some $r' \in grd(\Pi')$ exists such that $H(r') = \alpha$ and $X' \models B(r')$. Thus in particular for $\alpha \in U'$, every such rule r' must satisfy $B^+(r') \cap U' \neq \emptyset$. Inspecting the rules in Π', we can see that by construction of X' no bl_{n_r}-atoms and no ko_{n_r} atoms can occur in U', and $\overline{\alpha}$ atoms and ab_*-atoms only if U' contains some ap_{n_r}-atoms resp. atom α from HB_Π.

Now in the case $ap_{n_r}(c_1, \ldots, c_n) \in U'$, we must have some atom $\alpha \in B^+(r')$ in U' where r' has the form $ap_{n_r}(c_1, \ldots, c_n) \leftarrow B(r')$; similarly if $ap_{n_r}(d_1, \ldots, d_{n'}) \in U'$, we must have some atom $\alpha \in B^+(r')$ in U' where r' has the form $ap_{n_r}(d_1, \ldots, d_{n'}) \leftarrow B(r')$. In turn, if $\alpha \in U'$, we must have some atom $ap_{n_r}(c_1, \ldots, c_n) \in U'$ for r' of the form $\alpha \leftarrow ap_{n_r}(c_1, \ldots, c_n), not\, \overline{\alpha}$ where $\alpha = p(c_1, \ldots, c_n)$ (from the choice rule $\{H(r)\} \leftarrow ap_{n_r}(c_1, \ldots, c_n)$).

Continuing this argument, we obtain that each atom $\alpha \in U' \cap HB_\Pi$ is on a positive cycle in $G^+_{\Pi'}$. Since Π is positive-dependency founded, every rule $r \in \Pi$ that has an instance r' with $H(r') = \alpha$ must thus have $B^-(r) = \emptyset$ and $B^+(r) \neq \emptyset$. Define now the abstract interpretation $\hat{I}' = \hat{I} \setminus \{m(\alpha) \mid \alpha \in U' \cap HB_\Pi\}$ and consider Π^m. The abstraction of r introduces there only the rules (a)-(c) from Definition 3.4, but no rules with body shifts or literal removals. Hence, each instance r' of such a rule to derive $m(\alpha)$ for $\alpha \in U' \cap HB_\Pi$ has some atom $m(\beta)$ in $B^+(r)$ such that $\beta \in U' \cap HB_\Pi$. Consequently, $\hat{I}'$ satisfies $(\Pi^m)^{\hat{I}}$. However, this means that $\hat{I}$ is not a minimal model of $\Pi^{\hat{I}}$, which contradicts $\hat{I} \in AS(\Pi^m)$. Thus, an unfounded set U as supposed cannot exist. Consequently, X' is a minimal model of $\Pi'^{X'}$ and thus an answer set of Π', which proves the result. □

Proof of Proposition 4.3. If $\hat{I}$ is spurious, by Proposition 4.1 the program $\Pi \cup Q_{\hat{I}}^m$ is unsatisfiable. We focus on debugging the cause of inconsistency. Since Π is positive-dependency founded, this inconsistency can either be due to (i) an unsatisfied rule, (ii) an unsupported atom, or (iii) a supported atom that is on a positive cycle and has only positive atom dependencies. Indeed, if all rules are satisfied by Π and S is not an answer set of Π, then some unfounded set U of Π w.r.t. S exists such that $U \cap S \neq \emptyset$, cf. [75]; if all atoms in S are supported, then every atom $\alpha \in U \cap S$ is on a positive cycle in G^+_Π with nodes in $U \cap S$ and α has only positive atom dependencies as Π is positive-dependency founded. Consider now the three cases:

(i) Let $r \in \Pi$ be an unsatisfied rule w.r.t. S. This means that the constraints in $Q_{\hat{I}}^m$ is causing $H(r)$ to be false while $B(r)$ is satisfied. By the program $\mathcal{T}_{meta}[\Pi]$, depending on r, either $ap_{n_r}(c_1, \ldots, c_n)$ or $ap_{n_r}(d_1, \ldots, d_{n'})$ is true. By $\mathcal{T}_{deact}[\Pi]$, we get $ab\_deact_{n_r}(c_1, \ldots, c_n) \in S$. If $H(r) = \bot$, then by $\mathcal{T}_{deactCons}[\Pi]$, we have $ko_{n_r} \in S$ (else $\bot \in S$ by $\mathcal{T}_{meta}[\Pi]$), and we get $ab\_deactCons_{n_r}(d_1, \ldots, d_{n'}) \in S$.

(ii) Let $\alpha = p(c_1, \ldots, c_n) \in S$ be an unsupported atom in Π w.r.t. S for the domain elements $c_1, \ldots, c_n$. Then, for each rule instance r deriving α, we have $bl_{n_r}(c_1, \ldots, c_n) \in S$ and by $\mathcal{T}_{act}[\Pi, \mathcal{V}]$, we have $ab\_act_p(c_1, \ldots, c_n) \in S$.

(iii) Assuming all rules in Π are satisfied and all atoms in S are supported, we show that this case is not possible by deriving a contradiction. For some U as described, every instance r' of a rule $r : \alpha \leftarrow B(r)$ in Π such that $\alpha \in U \cap S$ and S satisfies $B(r')$ has $B^-(r') = \emptyset$, which means $B^-(r) = \emptyset$. Thus, the abstract program Π^m includes for r only rules without body-shifts, i.e., rules (a)–(c) in Definition 3.4. By positive-dependency foundedness, the interpretation $S \setminus (U \cap S)$ satisfies all rules in Π. Consequently, the abstract interpretation $\hat{I} \setminus m(U \cap S)$ satisfies the rules (a)–(c) constructed for r and is a model of Π^m; this means that $\hat{I}$ is not a minimal model of the GL-reduct of Π^m w.r.t. $\hat{I}$, which contradicts $\hat{I} \in AS(\Pi^m)$. □

Proof of Theorem 5.1. For an assignment I, we need to show that $I \cup \mathcal{T}_{m_\exists}$ is a minimal model of $(\Pi^m_\exists)^I$ if and only if $I \cup \mathcal{T}_m$ is a minimal model of $(\Pi^m)^I$.

($\Rightarrow$) Towards a contradiction, assume $I \cup \mathcal{T}_{m_\exists}$ is a minimal model of $(\Pi^m_\exists)^I$ but $I \cup \mathcal{T}_m$ is not a minimal model of $(\Pi^m)^I$. Then either (i) $I \cup \mathcal{T}_m$ is not a model of $(\Pi^m)^I$, or (ii) $I \cup \mathcal{T}_m$ is not a minimal model of $(\Pi^m)^I$.

(i) There is a rule $\hat{r} \in (\Pi^m)^I$ such that $I \cup \mathcal{T}_m \models B(\hat{r})$ but $I \not\models H(\hat{r})$. By construction of Π^m, $\hat{r}$ is only obtained by step (a) of Definition 3.4, otherwise $\hat{r}$ would be a choice rule with head $H(\hat{r}) = \{m(l)\}$, and $\hat{r}$ would be satisfied. Consequently $\hat{r}$ is a rule from step (a) for r in Π. Thus, we have $I \cup \mathcal{T}_m \models m(B^{std}(r)), \hat{d}_1 \circ \hat{d}_2, \tau_1^\circ(\hat{d}_1, \hat{d}_2)$. Since the definitions of relation type I for lifted relations and abstract relations correspond to each other, we have $\mathcal{T}_m \models \tau_1^\circ(\hat{d}_1, \hat{d}_2) \iff \mathcal{T}_{m_\exists} \models \tau_1^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$. This means we get $I \cup \mathcal{T}_{m_\exists} \models m(B^{std}(r)), \tau_1^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ which is the abstract rule of r constructed by step (a) of Definition 5.1. Since $I \cup \mathcal{T}_{m_\exists}$ is a minimal model of $(\Pi^m_\exists)^I$, $I \models H(\hat{r})$. Hence, we reach a contradiction.

(ii) Let there be $J \subset I$ such that $J \cup \mathcal{T}_m$ is a model of $(\Pi^m)^I$. We claim that $J \cup \mathcal{T}_{m_\exists}$ is a model of $(\Pi^m_\exists)^I$, which would contradict $I \cup \mathcal{T}_{m_\exists} \in AS(\Pi^m_\exists)$. Assume $J \cup \mathcal{T}_{m_\exists} \not\models (\Pi^m_\exists)^I$. Then there is a rule $\hat{r} \in (\Pi^m_\exists)^I$ such that $J \cup \mathcal{T}_{m_\exists} \models B(\hat{r})$ but $J \not\models H(\hat{r})$, while $I \models H(\hat{r})$. We need to show that there is a corresponding rule in $(\Pi^m)^I$ for $\hat{r}$, which would then achieve the contradiction that is $J \models H(\hat{r})$. Below, we denote by $B(\hat{r}) \setminus \hat{\Gamma}_r$, the abstract body excluding the abstracted relation (and its relation type atom).

- If $\hat{r}$ contains $\tau_{\mathrm{I}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ (step (a) or (c) of Definition 5.1), then since we know $\mathcal{T}_m \models \tau_{\mathrm{I}}^{\circ}(\hat{d}_1, \hat{d}_2) \iff \mathcal{T}_{m_\exists} \models \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{t}_1, \hat{t}_2)$, we achieve $J \cup \mathcal{T}_m \models \hat{d}_1 \circ \hat{d}_2$, $\tau_{\mathrm{I}}^{\circ}(\hat{d}_1, \hat{d}_2)$ (also $J \cup \mathcal{T}_m \models \hat{d}_1 \circ \hat{d}_2$). Since $\hat{r} \in (\Pi_\exists^m)^I$, we know that $I \models B(\hat{r})$ and also $I \models B(\hat{r}) \setminus \hat{\Gamma}_r$. Thus we get $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, \hat{d}_1 \circ \hat{d}_2, \tau_{\mathrm{I}}^{\circ}(\hat{d}_1, \hat{d}_2)$ (also $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, \hat{d}_1 \circ \hat{d}_2$) in $(\Pi^m)^I$. Since $J \cup \mathcal{T}_m$ is a model of $(\Pi^m)^I$, we get $J \models H(\hat{r})$, which is a contradiction.
- If $\hat{r}$ contains $\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ (step (b) or (c) of Definition 5.1), then $J \cup \mathcal{T}_{m_\exists} \models \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ means that there exist some $d_{11}, d_{12} \in m^{-1}(\hat{d}_1)$, $d_{21}, d_{22} \in m^{-1}(\hat{d}_2)$ and some $J' \in m^{-1}(J)$ such that $J' \models d_{11} \circ d_{21}$ and $J' \not\models d_{12} \circ d_{22}$. There are the following cases for $\hat{d}_1 \circ \hat{d}_2$: (1) $J \models \hat{d}_1 \circ \hat{d}_2$, or (2) $J \not\models \hat{d}_1 \circ \hat{d}_2$.

  (1) Since we know $J' \not\models d_{12} \circ d_{22}$, this case obtains $\tau_{\mathrm{III}}^{\circ}(\hat{d}_1, \hat{d}_2)$, thus $J \cup \mathcal{T}_m \models \hat{d}_1 \circ \hat{d}_2, \tau_{\mathrm{III}}^{\circ}(\hat{d}_1, \hat{d}_2)$. With similar reasoning as above on obtaining $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, \hat{d}_1 \circ \hat{d}_2, \tau_{\mathrm{III}}^{\circ}(\hat{d}_1, \hat{d}_2)$ in $(\Pi^m)^I$ (also $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, \hat{d}_1 \circ \hat{d}_2$ in $(\Pi^m)^I$), we achieve $J \models H(\hat{r})$, a contradiction.
  (2) Since we know $J' \models d_{11} \circ d_{21}$, this case obtains $\tau_{\mathrm{IV}}^{\circ}(\hat{d}_1, \hat{d}_2)$, thus $J \cup \mathcal{T}_m \models \hat{d}_1 \,\overline{\circ}\, \hat{d}_2, \tau_{\mathrm{IV}}^{\circ}(\hat{d}_1, \hat{d}_2)$. With similar reasoning as above we reach a contradiction.

($\Leftarrow$) Towards a contradiction, assume $I \cup \mathcal{T}_m$ is a minimal model of $(\Pi^m)^I$ but $I \cup \mathcal{T}_{m_\exists}$ is not a minimal model of $(\Pi_\exists^m)^I$. Then either (i) $I \cup \mathcal{T}_{m_\exists}$ is not a model of $(\Pi_\exists^m)^I$, or (ii) $I \cup \mathcal{T}_{m_\exists}$ is not a minimal model of $(\Pi_\exists^m)^I$.

(i) There is a rule $\hat{r} \in (\Pi_\exists^m)^I$ such that $I \cup \mathcal{T}_{m_\exists} \models B(\hat{r})$ but $I \not\models H(\hat{r})$. By construction of $\Pi_\exists^m$, $\hat{r}$ is only obtained by step (a) of Definition 5.1. With an analogous reasoning as above item (i), we achieve a contradiction.
(ii) Let there be $J \subset I$ such that $J \cup \mathcal{T}_{m_\exists}$ is a model of $(\Pi_\exists^m)^I$. We claim that $J \cup \mathcal{T}_m$ is a model of $(\Pi^m)^I$, which would contradict $I \cup \mathcal{T}_m \in AS(\Pi^m)$. Assume $J \cup \mathcal{T}_m \not\models (\Pi^m)^I$. Then there is a rule $\hat{r} \in (\Pi^m)^I$ such that $J \cup \mathcal{T}_m \models B(\hat{r})$ but $J \not\models H(\hat{r})$, while $I \models H(\hat{r})$. We need to show that there is a corresponding rule in $(\Pi_\exists^m)^I$ for $\hat{r}$, which would then achieve the contradiction that $J \models H(\hat{r})$.
  - If $\hat{r}$ contains $\hat{d}_1 \circ \hat{d}_2, \tau_{\mathrm{I}}^{\circ}(\hat{d}_1, \hat{d}_2)$ (step (a) of Definition 3.4), an analogous reasoning as above item (ii) obtains $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ in $(\Pi_\exists^m)^I$ which achieves $J \models H(\hat{r})$ a contradiction.
  - If $\hat{r}$ contains $\hat{d}_1 \circ \hat{d}_2, \tau_{\mathrm{III}}^{\circ}(\hat{d}_1, \hat{d}_2)$ (step (b) of Definition 3.4), then $J \cup \mathcal{T}_m \models \hat{d}_1 \circ \hat{d}_2, \tau_{\mathrm{III}}^{\circ}(\hat{d}_1, \hat{d}_2)$ means that $J \models \hat{d}_1 \circ \hat{d}_2$ and there exist some $d_1 \in m^{-1}(\hat{d}_1)$, $d_2 \in m^{-1}(\hat{d}_2)$ and some $J' \in m^{-1}(J)$ such that $J' \not\models d_1 \circ d_2$. This obtains abstract relation type $\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$, thus $J \cup \mathcal{T}_{m_\exists} \models \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$. Notice that also $J \models \mathit{not\ isSingleton}(\hat{d}_i)$ holds for some $i \in \{1, 2\}$. With similar reasoning as above on obtaining $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ in $(\Pi_\exists^m)^I$, we achieve $J \models H(\hat{r})$, a contradiction.
  - If $\hat{r}$ contains $\hat{d}_1 \,\overline{\circ}\, \hat{d}_2, \tau_{\mathrm{IV}}^{\circ}(\hat{d}_1, \hat{d}_2)$ (step (c) or (d.ii) of Definition 3.4), then $J \cup \mathcal{T}_{m_0} \models \hat{d}_1 \,\overline{\circ}\, \hat{d}_2, \tau_{\mathrm{IV}}^{\circ}(\hat{d}_1, \hat{d}_2)$ means that $J \not\models \hat{d}_1 \circ \hat{d}_2$ and there exist some $d_1 \in m^{-1}(\hat{d}_1)$, $d_2 \in m^{-1}(\hat{d}_2)$ and some $J' \in m^{-1}(J)$ such that $J' \models d_1 \circ d_2$. This again obtains abstract relation type $\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$, i.e., $J \cup \mathcal{T}_{m_\exists} \models \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$, thus reaches a contradiction as above.
  - If $\hat{r}$ contains only $\hat{d}_1 \circ \hat{d}_2$ (step (d.i) of Definition 3.4), then this means either $J \cup \mathcal{T}_{m_\exists} \models \tau_{\mathrm{I}}^{\circ}(\hat{d}_1, \hat{d}_2)$ or $J \cup \mathcal{T}_{m_\exists} \models \tau_{\mathrm{III}}^{\circ}(\hat{d}_1, \hat{d}_2)$ holds. Also we know that $J \models \mathit{not\ isSingleton}(\hat{d}_i)$ holds for some $i \in \{1, 2\}$. So similar as above, we achieve a contradiction. □

A.1. Merged vs. individual independent cycles

Let $\Pi_{L_{c_1}, \ldots, L_{c_l}}$ denote the program constructed in Definition 3.8 for $L_{c_1}, \ldots, L_{c_l}$, and let Π_S denote the program constructed by considering a big cycle $S = L_{c_1} \cup \ldots \cup L_{c_l}$ (i.e., using S in instead of L_{c_j} for defining $B^{sh}_{L,S}(r)$). Then we obtain the following result.

Proposition A.2. *For every program Π, it holds that $AS(\Pi_{L_{c_1}, \ldots, L_{c_l}}) = AS(\Pi_S)$.*

Proof. Let $\mathbf{L} = L_{c_1}, \ldots, L_{c_l}$. We first observe that the rules in $\Pi_{\mathbf{L}}$ and Π_S are related in the following way. We say that a rule r is a *tightening* of a rule r', if $H(r) = H(r')$, $B^+(r) \subseteq B^+(r')$ and $B^-(r) \subseteq B^-(r')$.

Lemma A.3. *For each rule $r \in \Pi_{\mathbf{L}}$ according to (d), there exists a tightening r' of r in Π_S, and for each rule $r \in \Pi_S$ according to (d), there exists a tightening r' of r in $\Pi_{\mathbf{L}}$.*

To see the first part, if $\alpha \in L_{c_j}$ then $\alpha \in S$, and if $\alpha \in L \cap L_{c_j}$ then $\alpha \in L \cap S$. Suppose a rule r in (d) is included in $\Pi_{\mathbf{L}}$ with $B^{sh}_{L,L_{c_j}}(r_0)$ for some rule $r_0 \in \Pi$. If in (42) the condition $\alpha \in L_{c_j}$ and $L_{c_j} \cap L \neq \emptyset$ applies, then a tightening r' of r will be included in Π_S for $B^{sh}_{L,S}(r_0)$ by condition $\alpha \in L_{c_j}$ and $L_{c_j} \cap L \neq \emptyset$; if in (42) the condition $\alpha \notin L_{c_j}$ or $L_{c_j} \cap L = \emptyset$ applies, then

likewise a tightening r' of r will be included in Π_S for $B^{sh}_{L,S}(r_0)$ by either the condition $\alpha \notin L_{c_j}$ or $L_{c_j} \cap L = \emptyset$ (then $(r = r')$ or by the condition $\alpha \in L_{c_j}$ and $L_{c_j} \cap L \neq \emptyset$.

For the second part, r stems from a rule $r_0 \in \Pi$ for S and some L. For each L_{c_j}, from r_0 some rule r' for L and L_{c_j} is added to $\Pi_{\mathbf{L}}$; If $r = r'$, we are done. Otherwise, it follows that $\alpha \in L_{c_j}$ and $L \cap L_{c_j} \neq \emptyset$ must hold.

Consider now the set $L' = L \setminus \bigcup_{j' \neq j} L_{c_{j'}}$. As each cycle $L_{c_{j'}}$, $j' \neq j$, is disjoint from L_{c_j}, we have $\alpha \in L_{c_j}$ and $L' \cap L_{c_j} \neq \emptyset$. Consequently, in (d) stemming from r_0 some rule r'' is put in $\Pi_{\mathbf{L}}$ for L' and L_{c_j} by the condition $\alpha \in L_{c_j}$ and $L_{c_j} \cap L \neq \emptyset$ in (42). Since $L' \setminus L_{c_i} = L \setminus S$ and $L' \subseteq L$, the rule r'' is a tightening of r. This proves the lemma.

The result follows now from this and the following lemma.

Lemma A.4. *Let Π_1 and Π_2 be programs that differ only on choice rules and such that, for each choice rule r in Π_1, some tightening r' of r in Π_2 exists. Then $AS(\Pi_1) \subseteq AS(\Pi_2)$.*

Indeed, consider $I \in AS(\Pi_1)$ and any choice rule $r' \in \Pi_2$. If r' is the tigthening of some rule $r \in \Pi_1$ such that $I \models B(r)$ and $I \models H(r)$, then we keep r for building an nswer set; otherwise, if $I \models B(r')$ we discard r'. Then by construction $I \models \Pi_2^I$, and since for each rule r in Π_1^I such that $I \models B(r)$ a tightening in Π_2^I exists, no smaller model J of Π_2^I is possible (otherwise I would not be the least model of Π_1^I). Thus, I is answer set of Π_2, which proves the lemma.

Putting things together, the programs $\Pi_{\mathbf{L}}$ and Π_S differ only on choice rules according to (d), and by Lemma A.3 the condition of Lemma A.4 is verified to conclude $AS(\Pi_{\mathbf{L}}) \subseteq AS(\Pi_S)$ and $AS(\Pi_S) \subseteq AS(\Pi_{\mathbf{L}})$, respectively, which completes the proof. □

Appendix B. Further details

B.1. Correctness checking failure for non-positive-dependency founded programs

The following example shows that the procedure for correctness checking does not work for programs that are not positive-dependency founded in general.

Example B.1. Consider the program below which is unsatisfiable and also contains a positive loop.

$$r_1 : a(X) \leftarrow not\ a(X), dom(X).$$
$$r_2 : a(X) \leftarrow a(X).$$
$$dom(1).\ dom(2).\ dom(3).$$

For the mapping $m = \{\{1,2,3\} \mapsto k\}$, the constructed abstract program Π^m is

$$a(X) \leftarrow not\ a(X), dom(X).$$
$$\{a(X)\} \leftarrow not\ isSingleton(X), dom(X).$$
$$a(X) \leftarrow a(X).$$
$$dom(k).$$

which has the abstract answer set $\hat{I} = \{a(k)\}$. Checking the correctness using $\Pi_{debug} \cup Q^m_{\hat{I}}$ results in unsatisfiability, because it requires to have some $a(c)$ for $c \in m^{-1}(k)$ to hold true through a loop, which is not covered in the definition of Π_{debug}. More in detail, the program Π_{debug} contains among others the following rules:

$$\{a(X)\} \leftarrow ap_{r_1}(X). \qquad \{a(X)\} \leftarrow ap_{r_2}(X).$$
$$ap_{r_1}(X) \leftarrow not\ a(X), dom(X). \qquad ap_{r_2}(X) \leftarrow a(X).$$
$$bl_{r_1}(X) \leftarrow not\ ap_{r_1}(X), dom(X). \qquad bl_{r_2}(X) \leftarrow not\ ap_{r_2}(X), dom(X).$$
$$\{a(X)\} \leftarrow bl_{r_1}(X), bl_{r_2}(X).$$

Then we may pick $X = \{a(1)\}$ (omitting the domain facts) as starting set for building an answer set X' of $\Pi' = \Pi_{debug} \cup Q^m_{\hat{I}}$ as in the proof of Proposition 4.2. Following the steps, we obtain $X' = \{a(1), ap_{r_2}(1), bl_{r_1}(1),\ ap_{r_1}(2), bl_{r_2}(2), \overline{a}(2), ap_{r_1}(3), bl_{r_2}(3), \overline{a}(3), \ldots\}$. This is not an answer set of Π', however, as the atoms $a(1)$, $ap_{r_2}(1)$ form a cycle in $G^+_{\Pi'}$ that is an unfounded set of Π' w.r.t. X'. Notice that $m(a(1)) = a(k)$ has in Π^m founded support from the choice rule, which allows for having $\hat{I}$ as answer set of Π^m. We would obtain a similar picture if we would choose any other $X \subseteq \mathcal{A}$ such that $m(X) = \hat{I}$ as a starting set, viz. that for each $c \in \{1,2,3\}$ such that $a(c) \in X$, we have $ap_{r_2}(c) \in X'$ for the X' constructed and $U = \{a(c), ap_{r_2}(c)\}$ is an unfounded set of Π^m w.r.t. X'.

B.2. Grid-cell problem encodings

In this appendix, we provide details about the slight modifications made from the existing (or common) encodings, in order to use them in our experiments. The full encodings can be found in www.kr.tuwien.ac.at/research/systems/abstraction/mdaspar_material.zip.

Sudoku We used the encoding from DLV group in ASPCOMP09 with slight modifications. The guessing of the assignment of numbers to the free cells is written as

$$\{sol(X, Y, N) : num(N)\} \leftarrow not\ occupied(X, Y), row(X), column(Y).$$
$$hasNum(X, Y) \leftarrow sol(X, Y, N).$$
$$\bot \leftarrow not\ hasNum(X, Y), row(X), column(Y).$$

The constraints of assigning one symbol per column and one symbol per row are the same as in the original encoding, but with standardizing apart over the sorts *row* and *column*.

$$\bot \leftarrow sol(X, Y_1, M), sol(X_2, Y_2, M), X = X_2, Y_1 < Y_2.$$
$$\bot \leftarrow sol(X_1, Y, M), sol(X_2, Y_2, M), X_1 < X_2, Y = Y_2.$$

For the constraint of assigning one symbol per subregion, standardizing apart the original rules caused to have relations with many arguments, thus we converted them into the rules

$$\bot \leftarrow sol(X_1, Y_1, M), sol(X_2, Y_2, M), sameSubSquareLessThan(X_1, Y_1, X_2, Y_2).$$
$$sameSubSquareLessThan(X_1, Y_1, X_2, Y_2) \leftarrow sameSubSquare(X_1, Y_1, X_2, Y_2), X_1 < X_2.$$
$$sameSubSquareLessThan(X_1, Y_1, X_2, Y_2) \leftarrow sameSubSquare(X_1, Y_1, X_2, Y_2), Y_1 < Y_2.$$
$$sameSubSquare(X_1, Y_1, X_2, Y_2) \leftarrow subrangeR(X_1, M), subrangeR(X_2, M), subrangeC(Y_1, R), subrangeC(Y_2, R).$$

with the hardcoded facts $subrangeR(X, M)$ and $subrangeC(Y, R)$ for subregions w.r.t. rows and columns, respectively.

Knight's tour We used the encoding from ASPCOMP11[12] with slight modifications. At most one *move* atom is made for each *valid* movement among the cells.

$$\{move(X_1, Y_1, X_2, Y_2)\}1 \leftarrow valid(X_1, Y_1, X_2, Y_2).$$

In the original encoding, the valid cells computations were done using rules of the form

$$valid(X_1, Y_1, X_2, Y_2) \leftarrow point(X_1, Y_1), point(X_2, Y_2), X_1 = X_2 + 2, Y_1 = Y_2 + 1.$$

which are modified as

$$validcell(X_1, Y_1, X_2, Y_2) \leftarrow dist1(X_1, X_2), dist2(Y_1, Y_2).$$
$$validcell(X_1, Y_1, X_2, Y_2) \leftarrow dist2(X_1, X_2), dist1(Y_1, Y_2).$$
$$valid(X_1, Y_1, X_2, Y_2) \leftarrow validcell(X_1, Y_1, X_2, Y_2), point(X_1, Y_1), point(X_2, Y_2).$$

where the auxiliary facts $dist1(X_1, X_2), dist2(X_1, X_2)$ represent the arithmetic operations $X_1 = X_2 + 2, Y_1 = Y_2 + 1$.

The constraints to ensure that exactly one entering/leaving movement is made for each cell is the same as the original encoding. Having each cell visited is ensured by the following rules

$$reached(X, Y) \leftarrow move(X_1, Y_1, X, Y), start(X_1, Y_1).$$
$$reached(X_2, Y_2) \leftarrow reached(X_1, Y_1), move(X_1, Y_1, X_2, Y_2).$$
$$\bot \leftarrow point(X, Y), not\ reached(X, Y), row(X), column(Y).$$

where the atom $start(X, Y)$ is used to show the starting point, instead of having in the rule the atom $move(1, 1, X, Y)$ as it is originally. This change makes treating the program more convenient, as the rules do not contain constants that need to mapped to different abstract constants depending on the mapping.

[12] www.mat.unical.it/aspcomp2011/files/KnightTour/knight_tour.enc.asp.

Visitall We encoded a planning problem along the guidelines in Section 7.2 on representing actions and change. We considered $go(X, Y, T)$ actions that can move horizontally/vertically to a cell X, Y. For such an action, we must ensure that the action does not pass through an obstacle or a previously visited cell, and that all the passed cells become visited. A common way of encoding this is to have auxiliary atoms that keep track of the cells that are in between, such as

$$aux\_passed(X, Y_2, T+1) \leftarrow rAt(X, Y, T), go(X, Y_1, T), Y < Y_2, Y_2 \leq Y_1.$$
$$aux\_passed(X, Y_2, T+1) \leftarrow rAt(X, Y, T), go(X, Y_1, T), Y_1 < Y_2, Y_2 \leq Y.$$
$$aux\_passed(X_2, Y, T+1) \leftarrow rAt(X, Y, T), go(X_1, Y, T), X < X_2, X_2 \leq X_1.$$
$$aux\_passed(X_2, Y, T+1) \leftarrow rAt(X, Y, T), go(X_1, Y, T), X_1 < X_2, X_2 \leq X.$$
$$passed(X, Y) \leftarrow aux\_passed(X, Y, T).$$

which are then used to ensure the above conditions.

$$\bot \leftarrow passed(X, Y), obsAt(X, Y).$$
$$visited(X, Y, T) \leftarrow aux\_passed(X, Y, T).$$
$$\bot \leftarrow aux\_passed(X, Y, T+1), visited(X, Y, T).$$

We follow the remark in Section 5 on handling different abstraction levels on variables in a rule. For example, for the first rule, in addition to the standardizing apart the rule as

$$aux\_passed(X, Y_2, T+1) \leftarrow rAt(X, Y, T), go(X_1, Y_1, T), X = X_1, Y < Y_2, Y_2 \leq Y_1.$$

we add the additional rule

$$aux\_passed(X_1, Y_2, T+1) \leftarrow rAt(X, Y, T), go(X_1, Y_1, T), X = X_1, Y < Y_2, Y_2 \leq Y_1.$$

We proceed similarly with the remaining rules.

Furthermore, knowing that the action $go(X_1, Y_1, T)$ will only be picked in a horizontal (resp. vertical) direction of $rAt(X, Y, T)$, we drop the relation $X = X_1$ (resp. $Y = Y_1$) from the body to make it smaller.

B.3. Example run of mDASPAR

To illustrate further considerations for debugging and refinement, we show an example run of mDASPAR.

Example B.2 *(ctd).* We run mDASPAR with the input program (Fig. 15) and the instance shown in Fig. 13, with the initial mapping m of clustering the grid-cell into four regions (Fig. 14a).

Step 1 After constructing the non-ground abstract program (Fig. 16) and computing the relation types, mDASPAR computes an abstract answer set

$$\{reachable(a_{1234}, b_{1234}), reachable(a_{5678}, b_{1234}), reachable(a_{1234}, b_{5678})\}.$$

Step 2 Correctness checking first uses Π_{debug_0} where the ab atoms only contain rule names (Fig. 27) to obtain the optimal answer set

$$\{ab\_deactCons_{r3}, ab\_deact_{r2}\}.$$

Step 3 Π_{debug} is constructed only for $r2, r3$ now respecting variables in the rules, and by defining $ab\_deact_{r2}(X_1, Y_1)$ as

$$ab\_deact_{r2}(X_1, Y_1) \leftarrow ap_{r2}(X_1, Y_1), not\ reachable(X_1, Y_1).$$
$$\bot :\sim ab\_deact_{r2}(X_1, Y_1).\ [1, r2, X_1, Y_1]$$
$$\bot :\sim ab\_deact_{r2}(X_1, Y_1), mapTo(X_1, Y_1, A_1, B_1), isSingleton(A_1), isSingleton(B_1).\ [20, r2, X_1, Y_1]$$
$$refine(A_1, B_1) \leftarrow ab\_deact_{r2}(X_1, Y_1), mapTo(X_1, Y_1, A_1, B_1), not\ isSingleton(A_1).$$
$$refine(A_1, B_1) \leftarrow ab\_deact_{r2}(X_1, Y_1), mapTo(X_1, Y_1, A_1, B_1), not\ isSingleton(B_1).$$

and similarly $ab\_deactCons_{r3}(X, Y, X_1, Y_1)$. Correctness checking finds an optimal answer set with the atoms

$$refine(a_{1234}, b_{5678}), refine(a_{5678}, b_{5678}).$$

$$
\begin{aligned}
ko_{r1}.&\\
\{reachable(X,Y)\} &\leftarrow ap_{r1}(X,Y).\\
ab\_deact_{r1} &\leftarrow ap_{r1}(X,Y), not\ reachable(X,Y).\\
\bot &:\sim ab\_deact_{r1}.\ [1, r1]\\
ko_{r2}.&\\
\{reachable(X_1,Y_1)\} &\leftarrow ap_{r2}(X_1,Y_1).\\
ab\_deact_{r2} &\leftarrow ap_{r2}(X_1,Y_1), not\ reachable(X_1,Y_1).\\
\bot &:\sim ab\_deact_{r2}.\ [1, r2]\\
ko_{r3}.&\\
ab\_deactCons_{r3} &\leftarrow ko_{r3}, ap_{r3}(X,Y,X_1,Y_1).\\
\bot &:\sim ab\_deactCons_{r3}.\ [1, r3]\\
ko_{r5}.&\\
\{neighbor(X,Y,X_1,Y_1)\} &\leftarrow ap_{r5}(X,Y,X_1,Y_1).\\
ab\_deact_{r5} &\leftarrow ap_{r5}(X,Y,X_1,Y_1), not\ neighbor(X,Y,X_1,Y_1).\\
\bot &:\sim ab\_deact_{r5}.\ [1, r5]\\
ko_{r6}.&\\
\{neighbor(X,Y,X_1,Y_1)\} &\leftarrow ap_{r6}(X,Y,X_1,Y_1).\\
ab\_deact_{r6} &\leftarrow ap_{r6}(X,Y,X_1,Y_1), not\ neighbor(X,Y,X_1,Y_1).\\
\bot &:\sim ab\_deact_{r6}.\ [1, r6]\\
&\\
\{neighbor(X,Y,X_1,Y_1)\} &\leftarrow bl_{r5}(X,Y,X_1,Y_1), bl_{r6}(X,Y,X_1,Y_1).\\
ab\_act(neighbor(X,Y,X_1,Y_1)) &\leftarrow bl_{r5}(X,Y,X_1,Y_1), bl_{r6}(X,Y,X_1,Y_1),\\
&\quad neighbor(X,Y,X_1,Y_1).\\
\bot &:\sim ab\_act_{neighbor}(X,Y,X_1,Y_1).\ [1, X, Y, X_1, Y_1]\\
&\\
\{reachable(X,Y)\} &\leftarrow bl_{r1}(X,Y), bl_{r2}(X,Y).\\
ab\_act(reachable(X,Y)) &\leftarrow bl_{r1}(X,Y), bl_{r2}(X,Y), reachable(X,Y).\\
\bot &:\sim ab\_act_{reachable}(X,Y).\ [1, X, Y]
\end{aligned}
$$

Fig. 27. Constructed debugging program $\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}]$.

Step 4 The region (a_{1234}, b_{5678}) is randomly picked to refine to $\{(a_{12}, b_{56}), (a_{12}, b_{78}), (a_{34}, b_{56}), (a_{34}, b_{78})\}$ by updating the corresponding mapping m.

Step 5 Relation types according to the new mapping are computed and the loop goes back to step 1 to compute a new abstract answer set.

The loop continues until unsatisfiability is achieved. The abstraction shown in Fig. 14b is one such abstraction where unsatisfiability is observed.

B.4. Modular concreteness checking

Ordered modularity An incremental approach to ASP solving proposed by Gebser et al. [53] builds on the concept of modules [94] and gradually increases the bound to the solution size, represented by a parameter k, to help with both grounding and solving. They are searching for an answer set with minimum size over k, thus they increment the parameter until an answer set is computed. We use a similar idea to detect the spuriousness of an abstract answer set by gradually increasing the parameter. However, in our case, the increment is continued until the spuriousness is realized with debugging, i.e., an answer set with an abnormality atom is obtained. We take a simpler view by limiting the generated grounding of the program to the parameter.

Let Π be a program with the Herbrand base $HB_\Pi = \mathcal{L}_B \cup \mathcal{L}_k$, for parameter k ranging over the natural numbers, where $\mathcal{L}_B$ represents the *static* literals with arguments independent of parameter k, and $\mathcal{L}_k$ represents the *dynamic* literals which have an argument k. For a set X of literals, we denote by $grd(\Pi)|_X = \{r \mid H(r) \cup B(r) \subseteq X\}$ the set of ground rules that contain only literals over X. Let $X_i \subseteq HB_\Pi$ denote the set of literals until the parameter value i, i.e., $X_i = \mathcal{L}_B \cup \bigcup_{j=0}^{i} \mathcal{L}_{k/j}$, where $\mathcal{L}_{k/j}$ denotes the set of literals with the respective argument of value j. The rules of $grd(\Pi)$ until parameter value i are then given by $grd(\Pi)|_{X_i}$, simply denoted $grd(\Pi)|_i$.

Let $I_{\leq i}$ denote the projection of an interpretation I to the literals related with the parameter value i, i.e., $I_{\leq i} = I \cap (\mathcal{L}_B \cup \bigcup_{0 \leq j \leq i} \mathcal{L}_i)$. We say that Π is *ordered modular*, if for each $I \in AS(\Pi)$ it holds that $I_{\leq i} \in AS(grd(\Pi)|_i)$ for all $0 \leq i \leq k$. We then know that determining the occurrence of a literal l in an answer set $I_{\leq i}$ relies only on the decisions made until point i.

Proposition B.1. *Let Π be an ordered modular program, m a domain mapping for Π, and let $\hat{I} \in AS(\Pi^m)$. If $\hat{I}_{\leq i} \subseteq \hat{I}$ is spurious for some $i \leq n$, then $\hat{I}$ is spurious.*

Proof. Assume $\hat{I}$ is concrete. This means that there exists some $I \in \Pi$ such that $m(I) = \hat{I}$. As Π is ordered modular, $I_{\leq i} \in AS(grd(\Pi)|_i)$. Thus, $m(I_{\leq i}) = \hat{I}_{\leq i}$ is concrete. □

We describe in detail the implementation of these approaches.

Incrementing *time*. Approach (1) is implemented in mDASPAR to handle planning problems with atoms having *time* arguments. By Proposition B.1, we know that if the first few actions of a potential plan described by the abstract answer set have no corresponding original plan, one can conclude that this plan is spurious.

A common description of the planning problem in ASP uses two sorts for time: $timea = \{0, \ldots, n-1\}$, which is used for action atoms, and $time = \{0, \ldots, n\}$ which is used to for the fluents. For a given program Π with a description of a planning problem that contains facts for sort *time*, mDASPAR works as follows. We denote by $\mathcal{T}_{meta}[\Pi]|_i$ the meta-program $\mathcal{T}_{meta}[\Pi]$ which contains *time* facts (resp. *timea*) until domain element i (resp. $i-1$), and by $\Pi_{debug}|_i$ the analogous restricted version of Π_{debug}. For a computed abstract answer set $\hat{I}$ encoding a plan $\langle s_0, a_0, s_1, \ldots, s_{n-1}, a_{n-1}, s_n \rangle$ we denote by $\hat{I}|_i$ the part of the plan until time point i.

Starting with $i = 1$, we continue the iteration below while $i \leq n$.

Step (1) Create $\mathcal{T}_{meta}[\Pi]|_{i-1}$ and $\Pi_{debug}|_i$.
Step (2) Check correctness of $\hat{I}|_i$ with $\Pi_{debug}|_i \cup \mathcal{T}_{meta}[\Pi]|_{i-1} \cup Q^m_{\hat{I}|_i}$.
Step (3) If $\hat{I}|_i$ is spurious, exit loop; otherwise, increase i by 1.

This way, we check the correctness of $\hat{I}$ for the action taken at time i, by debugging only for time point i as the guessing for time points $t < i$ is restricted by using $\mathcal{T}_{meta}[\Pi]|_{i-1}$. The time is increased incrementally while the partial solution yields a concrete partial plan. Once spuriousness is observed, the checking is stopped.

Partial concretization. For approach (2), we use the possibility to have a hierarchy of abstractions mentioned in Proposition 3.6. The idea is to partially concretize the abstract domain, by fully concretizing certain regions and keeping the remaining ones abstract. Fig. 17 shows the hierarchy of some partial concretizations of the initial mapping. For a given mapping m, we consider a set of possible partial concretizations. We then check the correctness of an abstract answer set I over the program with partially concretized domain. As the latter still describes an abstraction compared to the original domain, this check cannot be immediately done over the original program. For that, we have must check correctness with debugging over the abstract program relative to the partial concretization.

The approach is implemented in mDASPAR as follows. For a given mapping m, starting with $j = 1$, the iteration focuses on concretizing j regions at a time, and checks the correctness in each such j-region combination. The iteration continues until spuriousness is detected or $m = m_{id}$:

step (1) Compute j-region concretizations of m, say $m_1, \ldots, m_n$.
step (2) For every $m_i \in \{m_1, \ldots, m_n\}$;

1. Create Π^{m_i} with $\mathcal{T}_{m_i}$ and the set $\{m_i(p(c)). \mid p(c). \in \Pi\}$ of facts, and $\Pi^{m_i}_{debug}$.
2. Create the mapping m' such that $m'(m_i(D)) = m(D)$.
3. Check correctness of I with $\Pi^{m_i}_{debug} \cup Q^{m'}_I$.
4. If spurious, exit loop with debug answer C.

step (3) If $C \neq \emptyset$, refine m according to C and go back to step (1); otherwise, increase j by 1, and go back to step (1)

We do correctness checking on the abstract level m_i using Π^{m_i}. If I is concrete w.r.t. the partially concretized abstraction, this does not guarantee that I is concrete; thus, the concretization is increased to redo the check. If spuriousness is detected, the mapping is refined and the partial concretization continues from the updated mapping.

Appendix C. Use case: abstraction for policy refutation

As a further example, consider checking whether an agent always manages to find a missing person with a given policy in a grid environment with obstacles. If the policy does not work, a counterexample trajectory over some part of the environment will reveal this; by inspecting the latter, one may guess why it fails. Depending on the problem, the focus points may have different nature. For the reachability example shown in Section 5.2, the focus area in the environment can remain local, while for the person search example the path of a trajectory needs to be distinguished.

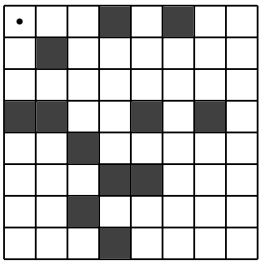
For illustration, we use the following running example.

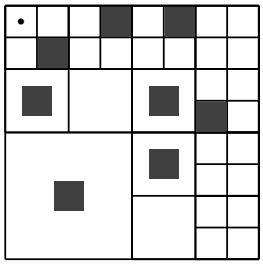
Example C.1 *(Example 5.8 cont'd).* Consider the reachability problem described in the following encoding, where reachability ((72)-(74)) is redefined by prioritizing the east neighbor over the rest, and in case the east neighbor has an obstacle, choosing the south neighbor.

$$point(X, Y) \leftarrow not\ obsAt(X, Y), row(X), column(Y). \tag{102}$$

$$reachable(X, Y) \leftarrow start(X, Y). \tag{103}$$

(a) Original domain (b) Distinguishing the cells reachable under the restrictions





$start(a_1, b_1)$. $obsAt(a_4, b_1)$.
$obsAt(a_6, b_1)$. $obsAt(a_2, b_2)$.
$obsAt(a_7, b_4)$. $obsAt(a_{12}, a_{34})$.
$obsAt(a_{56}, a_{34})$. $obsAt(a_{56}, a_{56})$.
$obsAt(a_{1234}, a_{5678})$.

Fig. 28. Reachability abstraction under east neighbor prioritization.

$$neighborE(X, Y, X_1, Y) \leftarrow X_1 - X = 1, column(Y). \tag{104}$$

$$neighborS(X, Y, X, Y_1) \leftarrow Y_1 - Y = 1, row(X). \tag{105}$$

$$reachableE(X, Y, X_1, Y) \leftarrow reachable(X, Y), point(X_1, Y), neighborE(X, Y, X_1, Y). \tag{106}$$

$$hasNeighborE(X, Y) \leftarrow reachableE(X, Y, X_1, Y_1). \tag{107}$$

$$reachable(X_1, Y_1) \leftarrow reachableE(X, Y, X_1, Y_1). \tag{108}$$

$$reachable(X, Y_1) \leftarrow reachable(X, Y), point(X, Y_1), neighborS(X, Y, X, Y_1), not\ hasNeighborE(X, Y). \tag{109}$$

For the instance shown in Fig. 28a the reachable cells are determined in the order $\rightarrow^2\downarrow\rightarrow^5\downarrow^6$. The abstraction shown in Fig. 28b singles out the area that contains the cells that are reachable according to the restrictions.

We now focus on using the abstraction over grid-cells for the problem of checking policies on whether they manage to guide the agent towards the goal. We consider two versions of this problem and discuss the use of abstraction.

As a running example, we consider an agent trying to find its way in a maze towards a goal point (similar in spirit to the example of finding a missing person). For representing and generating the mazes, we use an altered version of the Maze Generation encoding from ASP Competition 2011.[13] A policy that may come to one's mind when talking about mazes is the well-known "right-hand rule", which is known to work in many maze instances, except when the goal is in the middle area and the agent is forced to loop due to the obstacle layout.

Does the policy work on a given instance? For fixed problem instances, this check is done by a search of a counterexample trajectory which follows the policy but does not reach the goal. If none is found (i.e., unsatisfiability is achieved), we conclude that the policy works for this instance. Abstraction can be used to focus on the part of the instance which is enough to show that the policy fails or works; notice that the latter case becomes similar to having unsatisfiable problems. The necessary granularity of the abstraction depends on the complexity of the policy. As demonstrated in Fig. 30, for refuting the "right-hand rule" policy the abstraction must refine at least the outer area (if not more).

To observe how the policy type affects the resulting abstraction, we did some experiments. To help with the refinement decisions, the initial abstraction distinguishes the starting point of the agent and abstracts over the rest.

We consider the following two policies:

(A) Right-hand rule: Follow the wall on the right-hand side.
(B) Naive policy: Choose the direction to move to with the priority right > down > left > up.

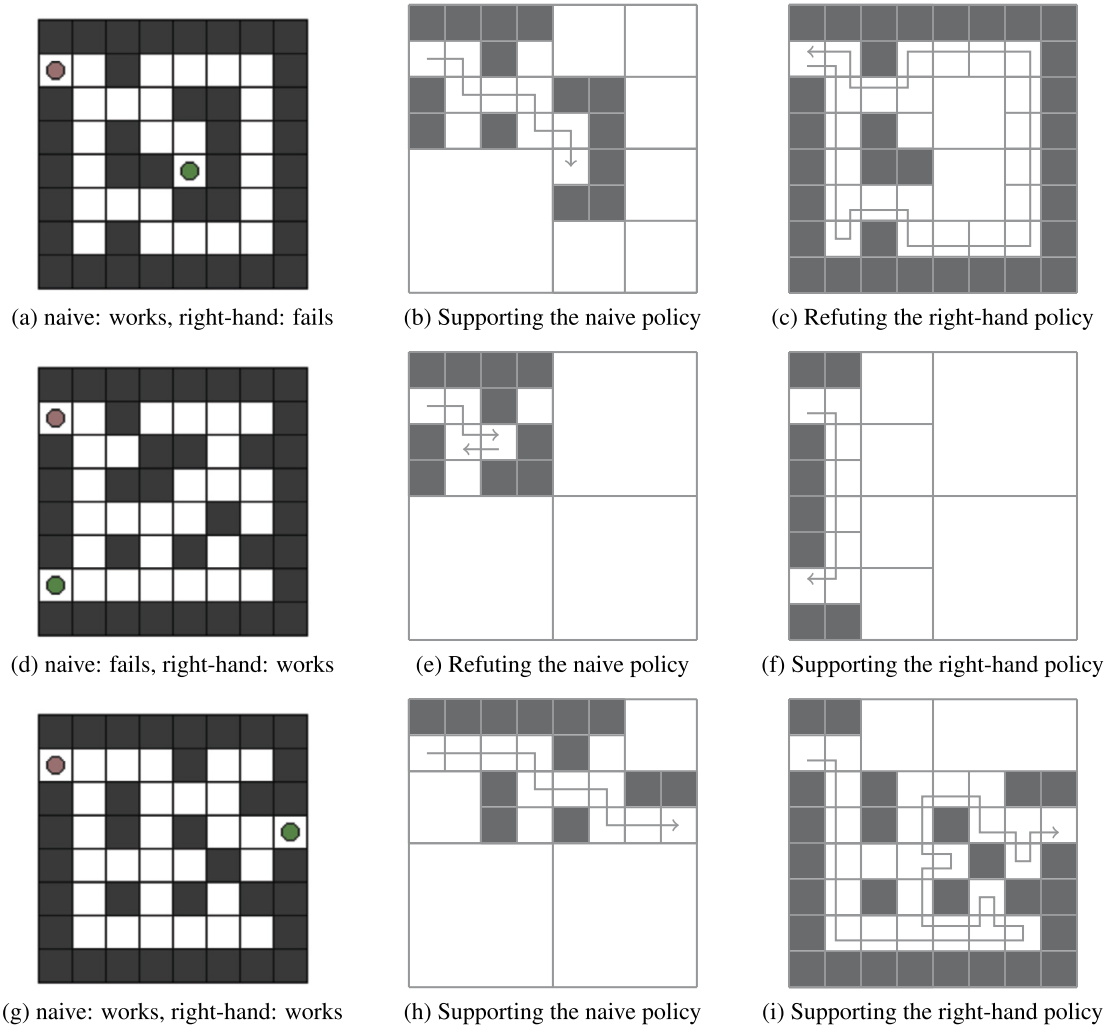
We generated 20 instances where on some of them both, one, or none of the policies work. For the debugging method we picked time increment, as we wanted it to focus on each step of the abstract trajectory starting from the beginning, and on whether or not the steps match the policy's decisions in the corresponding original trajectory. Furthermore, the refinement decision is made only from one abstract answer set, to avoid that a concrete answer set is encountered among spurious ones; this would finalize the search and achieve a clearly non-faithful abstraction.

Table 7 shows the results of using mDASPAR to achieve an abstraction with a concrete solution. Obtaining SAT means that the program found a concrete solution, i.e., a concrete counterexample trajectory, which shows that the policy does not work, while having UNSAT means that the policy works. As expected, the naive policy failed to work for most of the

[13] https://www.mat.unical.it/aspcomp2011/files/MazeGeneration/maze_generation.enc.asp.

Table 7
Policy checking in maze instances.

	sat/unsat	avg. step	avg. cost	avg. best step	avg. best cost
naive	16/4	7.2	0.391	6.5	0.362
right-hand	6/14	12.5	0.630	11.8	0.599



(a) naive: works, right-hand: fails (b) Supporting the naive policy (c) Refuting the right-hand policy

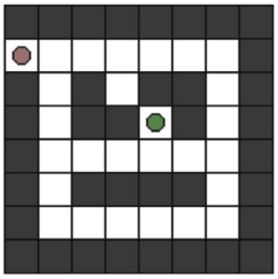
(d) naive: fails, right-hand: works (e) Refuting the naive policy (f) Supporting the right-hand policy

(g) naive: works, right-hand: works (h) Supporting the naive policy (i) Supporting the right-hand policy

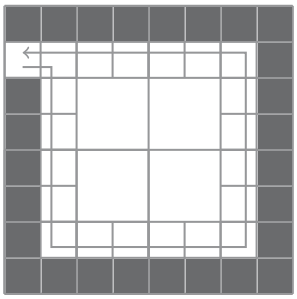
Fig. 29. Abstractions on policy checking in maze instances (with the supporting/refuting paths).

instances. Since the right-hand rule forces to traverse the environment more, mDASPAR required to have finer abstractions to figure out the concrete solution. In both cases, the obtained abstractions were not too distant from the best possible ones, although still sometimes the focus was shifted to the irrelevant parts of the grid. All of the obtained resulting abstractions were faithful, which means that they were able to show the actual behavior of the policy. Fig. 29 shows the resulting abstractions for three of the instances.

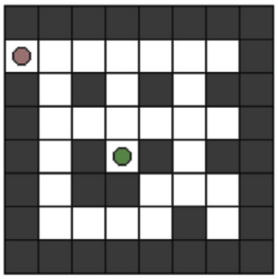
Does the policy always work? This is a more involved check, since a set of possible instances has to be considered and a search of a counterexample trajectory among each instance needs to be done. If the policy works, then all possible policy trajectories in all instances have to be checked to conclude this result. For this case, considering an abstraction that focuses on a certain part of the grid may not be useful, since depending on the structure of the instances different parts of the grid may need to be singled out. However, if the policy does not work, it is enough to find an instance in which a counterexample policy trajectory can be shown. Thus, an abstraction that focuses on a certain part of the grid where some instance can show a counterexample would be useful.



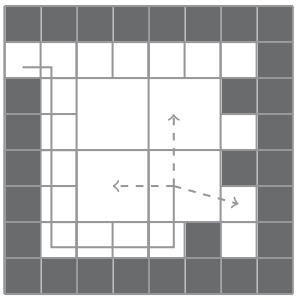
(a) A counterexample instance



(b) Distinguishing the path of a counterexample trajectory



(c) A counterexample instance



(d) Spurious counterexample trajectories due to abstract regions

Fig. 30. Can we refute the right-hand rule policy in all maze instances with one abstraction?

In ASP, such a check can be done by making two sets of guesses: (1) choose a valid instance, by guessing the layout of the environment and the position of the goal, and (2) determine a counterexample trajectory, by guessing the movements of the agent following the policy which do not achieve the goal in the instance. If the policy is deterministic (i.e., chooses exactly one action at a state), then the second guessing part becomes straightforward. However, for nondeterministic policies, a choice of possible actions to take exists, which adds to the complexity of the search.

The experiments showed that combining these guesses and those introduced in the syntactic transformation causes many spurious abstract answer sets, which sometimes force refinement decisions towards useless parts of the grid. For example, in general policy checking for the right-hand rule, mDASPAR must go back to the original domain to catch an instance with a counterexample trajectory, as the policy forces to traverse the environment and in the abstract encoding the guesses of the instance and the movements cause to create many spurious trajectories. As for the naive policy, mDASPAR can encounter a counterexample trajectory in few refinement steps: it is sufficient to realize that this policy fails by creating a partial instance where the agent enters a dead-end and has to leave by moving left; it then starts looping by moving right and left.

We remark that for a failing policy we may not expect to have one abstraction mapping that can be applied with any possible instance and catch a counterexample trajectory, the less a mapping that is faithful for any instance. Fig. 30a shows an instance in which the right-hand policy is unable to reach the (green) goal point from the (red) entry point in the upper left corner. An abstraction such as Fig. 30b is enough to realize that a loop occurs and a goal cannot be reached (it is a faithful abstraction for this instance). However, this abstraction does not always distinguish the cells that force to obtain a counterexample trajectory in each possible refuting instance. For example, the instance in Fig. 30c also forces the agent to loop; since with the same abstraction (Fig. 30d) there is uncertainty among the abstract regions, it is still possible to create spurious counterexample trajectories. Thus, faithfulness cannot be achieved. Here the identity abstraction would be the one that can be used to (faithfully) refute the policy in all possible instances.

References

[1] M. Alviano, C. Dodaro, Anytime answer set optimization via unsatisfiable core shrinking, Theory Pract. Log. Program. 16 (2016) 533–551.
[2] M. Alviano, C. Dodaro, M. Järvisalo, M. Maratea, A. Previti, Cautious reasoning in ASP via minimal models and unsatisfiable cores, Theory Pract. Log. Program. 18 (2018) 319–336, https://doi.org/10.1017/S1471068418000145.
[3] M. Alviano, W. Faber, N. Leone, Disjunctive ASP with functions: decidable queries and effective computation, Theory Pract. Log. Program. 10 (2010) 497–512, https://doi.org/10.1017/S1471068410000244.
[4] J.S. Anderson, A.M. Farley, Plan abstraction based on operator generalization, in: Proceedings of the 7th National Conference on Artificial Intelligence, AAAI 1988, 1988, pp. 100–104.
[5] B. Andres, B. Kaufmann, O. Matheis, T. Schaub, Unsatisfiability-based optimization in clasp, in: Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, Schloss Dagstuhl-Leibniz-Zentrum Fuer Informatik, 2012, pp. 211–221.
[6] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, Theory Pract. Log. Program. 18 (2018) 337–354.
[7] F. Bacchus, Q. Yang, Downward refinement and the efficiency of hierarchical problem solving, Artif. Intell. 71 (1994) 43–100.

[8] C. Backstrom, P. Jonsson, Planning with abstraction hierarchies can be exponentially less efficient, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence, vol. 2, IJCAI 1995, Morgan Kaufmann Publishers Inc., 1995, pp. 1599–1604.
[9] B. Banihashemi, G. De Giacomo, Y. Lespérance, Abstraction in situation calculus action theories, in: Proceedings of the 31st AAAI Conference on Artificial Intelligence, AAAI 2017, 2017, pp. 1048–1055.
[10] B. Banihashemi, G. De Giacomo, Y. Lespérance, Abstraction of agents executing online and their abilities in the situation calculus, in: Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI 2018, 2018, pp. 1699–1706.
[11] F. Belardinelli, A. Lomuscio, J. Michaliszyn, Agent-based refinement for predicate abstraction of multi-agent systems, in: Proceedings of the 22nd European Conference on Artificial Intelligence, ECAI 2016, IOS Press, 2016, pp. 286–294.
[12] M. Bichler, M. Morak, S. Woltran, The power of non-ground rules in answer set programming, Theory Pract. Log. Program. 16 (2016) 552–569.
[13] S. Bistarelli, P. Codognet, F. Rossi, Abstracting soft constraints: framework, properties, examples, Artif. Intell. 139 (2002) 175–211.
[14] B. Bonet, G. De Giacomo, H. Geffner, S. Rubin, Generalized planning: non-deterministic abstractions and trajectory constraints, in: Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI 2017, 2017, pp. 873–879.
[15] B. Bonet, H. Geffner, Policies that generalize: solving many planning problems with the same policy, in: Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, AAAI Press, 2015.
[16] M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, S. Woltran, Debugging asp programs by means of asp, in: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2007, Springer, 2007, pp. 31–43.
[17] S. Brass, J. Dix, Characterizations of the disjunctive stable semantics by partial evaluation, J. Log. Program. 32 (1997) 207–228.
[18] G. Brewka, T. Eiter, M. Truszczyński, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103.
[19] G. Brewka, T. Eiter, M. Truszczyński (Eds.), Answer Set Programming, AI Mag. 37 (3) (2016), AAAI Press.
[20] P. Cabalar, J. Fandinno, M. Fink, Causal graph justifications of logic programs, Theory Pract. Log. Program. 14 (2014) 603–618.
[21] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-core-2 input language format, arXiv:1911.04326, 2019.
[22] B.D. Cat, M. Denecker, M. Bruynooghe, P.J. Stuckey, Lazy model expansion: interleaving grounding with search, J. Artif. Intell. Res. 52 (2015) 235–286, https://doi.org/10.1613/jair.4591.
[23] L. Chittaro, R. Ranon, Hierarchical model-based diagnosis based on structural abstraction, Artif. Intell. 155 (2004) 147–182.
[24] K.L. Clark, Negation as failure, in: Logic and Data Bases, Springer, 1978, pp. 293–322.
[25] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, J. ACM 50 (2003) 752–794.
[26] E.M. Clarke, O. Grumberg, D.E. Long, Model checking and abstraction, ACM Trans. Program. Lang. Syst. (1994) 1512–1542.
[27] E.M. Clarke, T.A. Henzinger, H. Veith, R. Bloem, Handbook of Model Checking, Springer, 2018.
[28] M. Cohen, M. Dam, A. Lomuscio, F. Russo, Abstraction in model checking multi-agent systems, in: International Conference on Autonomous Agents and Multiagent Systems, vol. 2, 2009, pp. 945–952.
[29] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, J. Log. Program. 13 (1992) 103–179.
[30] K.J.W. Craik, The Nature of Explanation, vol. 445, CUP Archive, 1952.
[31] J.C. Culberson, J. Schaeffer, Pattern databases, Comput. Intell. 14 (1998) 318–334.
[32] D. Dams, R. Gerth, O. Grumberg, Abstract interpretation of reactive systems, ACM Trans. Program. Lang. Syst. 19 (1997) 253–291.
[33] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (2001) 374–425.
[34] J. Devriendt, B. Bogaerts, M. Bruynooghe, M. Denecker, On local domain symmetry for model expansion, Theory Pract. Log. Program. 16 (2016) 636–652, https://doi.org/10.1017/S1471068416000508.
[35] J. Dix, U. Kuter, D. Nau, Planning in answer set programming using ordered task decomposition, in: Annual Conference on Artificial Intelligence, Springer, 2003, pp. 490–504.
[36] C. Dodaro, P. Gasteiger, B. Musitsch, F. Ricca, K. Shchekotykhin, Interactive debugging of non-ground asp programs, in: Logic Programming and Nonmonotonic Reasoning, LPNMR, Springer, 2015, pp. 279–293.
[37] C. Drescher, O. Tifrea, T. Walsh, Symmetry-breaking answer set solving, AI Commun. 24 (2011) 177–194, https://doi.org/10.3233/AIC-2011-0495.
[38] S. Edelkamp, Planning with pattern databases, in: Proceedings of the 6th European Conference on Planning, ECP 2001, 2001, pp. 13–24.
[39] 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 (2007) 123.
[40] T. Eiter, M. Fink, Uniform equivalence of logic programs under the stable model semantics, in: International Conference on Logic Programming, Springer, 2003, pp. 224–238.
[41] T. Eiter, M. Fink, T. Krennwallner, C. Redl, HEX-programs with existential quantification, in: M. Hanus, R. Rocha (Eds.), Declarative Programming and Knowledge Management - Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11–13, 2013, Springer, 2013, pp. 99–117, Revised Selected Papers, https://doi.org/10.1007/978-3-319-08909-6_7.
[42] T. Eiter, M. Fink, H. Tompits, S. Woltran, Simplifying logic programs under uniform and strong equivalence, in: V. Lifschitz, I. Niemelä (Eds.), Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2004, Springer, 2004, pp. 87–99.
[43] T. Eiter, M. Fink, S. Woltran, Semantical characterizations and complexity of equivalences in answer set programming, ACM Trans. Comput. Log. 8 (2007) 17, https://doi.org/10.1145/1243996.1244000.
[44] T. Eiter, G. Gottlob, Y. Gurevich, Normal forms for second-order logic over finite structures, and classification of NP optimization problems, Ann. Pure Appl. Log. 78 (1996) 111–125.
[45] T. Eiter, Z.G. Saribatur, P. Schüller, Abstraction for zooming-in to unsolvability reasons of grid-cell problems, in: Proceedings of the IJCAI 2019 Workshop on Explainable Artificial Intelligence (XAI), 2019, pp. 7–13, Online available at https://drive.google.com/file/d/1ma5wilaj31A0d5KC4I2fYaTC_Lqm_d9X/view and http://arxiv.org/abs/1909.04998.
[46] T. Eiter, H. Tompits, S. Woltran, On solution correspondences in answer-set programming, in: L.P. Kaelbling, A. Saffiotti (Eds.), IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005, Professional Book Center, 2005, pp. 97–102, http://ijcai.org/Proceedings/05/Papers/1177.pdf.
[47] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68.
[48] E. Erdem, V. Patoglu, Applications of ASP in robotics, Künstl. Intell. 32 (2018) 143–149.
[49] A.A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E.C. Teppan, Industrial applications of answer set programming, Künstl. Intell. 32 (2018) 165–176.
[50] J. Fandinno, C. Schulz, Answering the "why" in answer set programming - a survey of explanation approaches, Theory Pract. Log. Program. 19 (2019) 114–203.
[51] M. Fox, D. Long, The detection and exploitation of symmetry in planning problems, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI 1999, 1999, pp. 956–961.
[52] E.C. Freuder, Eliminating interchangeable values in constraint satisfaction problems, in: AAAI, 1991, pp. 227–233.
[53] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, S. Thiele, Engineering an incremental ASP solver, in: Proceedings of the 24th International Conference on Logic Programming, ICLP 2008, 2008, pp. 190–205.
[54] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, Advanced preprocessing for answer set solving, in: Proceedings of the 18th European Conference on Artificial Intelligence, ECAI 2008, IOS Press, 2008, pp. 15–19.

[55] M. Gebser, J. Pührer, T. Schaub, H. Tompits, A meta-programming technique for debugging answer-set programs, in: Proceedings of the 23rd AAAI Conference on Artificial Intelligence, AAAI 2008, 2008, pp. 448–453.
[56] T. Geibinger, H. Tompits, Characterising relativised strong equivalence with projection for non-ground answer-set programs, in: F. Calimeri, N. Leone, M. Manna (Eds.), Logics in Artificial Intelligence - 16th European Conference, Proceedings, JELIA 2019, Rende, Italy, May 7-11, 2019, Springer, 2019, pp. 542–558.
[57] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (1991) 365–385.
[58] E. Giunchiglia, Y. Lierler, M. Maratea, Sat-based answer set programming, in: Proceedings of the 19th National Conference on Artificial Intelligence, AAAI 2004, 2004, pp. 61–66.
[59] F. Giunchiglia, T. Walsh, A theory of abstraction, Artif. Intell. 57 (1992) 323–389.
[60] G. Gottlob, N. Leone, H. Veith, Succinctness as a source of complexity in logical formalisms, Ann. Pure Appl. Log. 97 (1999) 231–260.
[61] M. Helmert, P. Haslum, J. Hoffmann, et al., Flexible abstraction heuristics for optimal sequential planning, in: Proceedings of the 17th International Conference on Automated Planning and Scheduling, ICAPS 2007, 2007, pp. 176–183.
[62] I.T. Hernádvölgyi, R.C. Holte, Psvn: A Vector Representation for Production Systems, 1999.
[63] J. Hoffmann, A. Sabharwal, C. Domshlak, Friends or foes? An AI planning perspective on abstraction and search, in: Proceedings of the 16th International Conference on Automated Planning and Scheduling, ICAPS 2006, 2006, pp. 294–303.
[64] R.C. Holte, T. Mkadmi, R.M. Zimmer, A.J. MacDonald, Speeding up problem solving by abstraction: a graph oriented approach, Artif. Intell. 85 (1996) 321–361.
[65] L. Illanes, S.A. McIlraith, Numeric planning via search space abstraction, in: Proceedings of the Workshop on Knowledge-Based Techniques for Problem Solving and Reasoning, 2016.
[66] L. Illanes, S.A. McIlraith, Generalized planning via abstraction: arbitrary numbers of objects, in: Proceedings of the 33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 2019.
[67] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, J.H. You, Unfolding partiality and disjunctions in stable model semantics, ACM Trans. Comput. Log. 7 (2006) 1–37.
[68] P.N. Johnson-Laird, Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness, Harvard University Press, 1983.
[69] S. Kambhampati, L. Davis, Multiresolution path planning for mobile robots, IEEE J. Robot. Autom. 2 (1986) 135–145.
[70] C.A. Knoblock, Automatically generating abstractions for planning, Artif. Intell. 68 (1994) 243–302.
[71] J. Kramer, Is abstraction the key to computing?, Commun. ACM 50 (2007) 36–42.
[72] C. Lefèvre, C. Béatrix, I. Stéphan, L. Garcia, Asperix, a first-order forward chaining approach for answer set computing, Theory Pract. Log. Program. 17 (2017) 266–310.
[73] J. Leite, A bird's-eye view of forgetting in answer-set programming, in: Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2017, Springer, 2017, pp. 10–22.
[74] 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 (2006) 499–562.
[75] N. Leone, P. Rullo, F. Scarcello, Disjunctive stable models: unfounded sets, fixpoint semantics, and computation, Inf. Comput. 135 (1997) 69–112.
[76] M.H. Liffiton, K.A. Sakallah, Algorithms for computing minimal unsatisfiable subsets of constraints, J. Autom. Reason. 40 (2008) 1–33.
[77] V. Lifschitz, Answer set planning, in: International Conference on Logic Programming, ICLP, 1999, pp. 23–37.
[78] V. Lifschitz, Twelve definitions of a stable model, in: M. Garcia de la Banda, E. Pontelli (Eds.), Logic Programming, Springer, Berlin Heidelberg, 2008, pp. 37–51.
[79] V. Lifschitz, What is answer set programming?, in: Proceedings of the 23rd AAAI Conference on Artificial Intelligence, AAAI 2008, 2008, pp. 1594–1597.
[80] V. Lifschitz, D. Pearce, A. Valverde, Strongly equivalent logic programs, ACM Trans. Comput. Log. 2 (2001) 526–541.
[81] F. Lin, Y. Zhao, Assat: computing answer sets of a logic program by sat solvers, Artif. Intell. 157 (2004) 115–137.
[82] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, D. Probst, Property preserving abstractions for the verification of concurrent systems, Form. Methods Syst. Des. 6 (1995) 11–44.
[83] A. Lomuscio, J. Michaliszyn, An abstraction technique for the verification of multi-agent systems against atl specifications, in: Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning, KR 2014, AAAI Press, 2014.
[84] A. Lomuscio, J. Michaliszyn, Verification of multi-agent systems via predicate abstraction against ATLK specifications, in: Proceedings of AAMAS, 2016, pp. 662–670.
[85] I. Lynce, J.P.M. Silva, On computing minimum unsatisfiable cores, in: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT 2004, 2004.
[86] M.J. Maher, Equivalences of logic programs, in: E. Shapiro (Ed.), Third International Conference on Logic Programming, Springer, Berlin Heidelberg, 1986, pp. 410–424.
[87] K. Marple, E. Salazar, G. Gupta, Computing stable models of normal logic programs without grounding, arXiv:1709.00501, 2017.
[88] M. Morak, S. Woltran, Preprocessing of complex non-ground rules in answer set programming, in: Technical Communications of the 28th International Conference on Logic Programming, 2012, p. 247.
[89] I. Mozetič, Hierarchical model-based diagnosis, Int. J. Man-Mach. Stud. 35 (1991) 329–362.
[90] P.P. Nayak, A.Y. Levy, A semantic theory of abstractions, in: Proc. International Joint Conference on Artificial Intelligence, 1995, pp. 196–203.
[91] B. Nebel, Y. Dimopoulos, J. Koehler, Ignoring irrelevant facts and operators in plan generation, in: European Conference on Planning, Springer, 1997, pp. 338–350.
[92] A. Newell, H.A. Simon, Human Problem Solving, Prentice-Hall, 1972.
[93] J. Oetsch, J. Pührer, H. Tompits, Catching the ouroboros: on debugging non-ground answer-set programs, Theory Pract. Log. Program. 10 (2010) 513–529.
[94] E. Oikarinen, T. Janhunen, Modular equivalence for normal logic programs, in: Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006, IOS Press, 2006, pp. 412–416.
[95] M. Osorio, J.A. Navarro, J. Arrazola, Equivalence in answer set programming, in: A. Pettorossi (Ed.), Logic Based Program Synthesis and Transformation, Springer, Berlin Heidelberg, 2002, pp. 57–75.
[96] A.D. Palù, A. Dovier, E. Pontelli, G. Rossi, Gasp: answer set programming with lazy grounding, Fundam. Inform. 96 (2009) 297–322.
[97] D. Pearce, Simplifying logic programs under answer set semantics, in: Logic Programming, 2004, pp. 210–224.
[98] D. Pearce, Equilibrium logic, Ann. Math. Artif. Intell. 47 (2006) 3–41.
[99] D. Pearce, A. Valverde, Synonymus theories in answer set programming and equilibrium logic, in: R.L. de Mántaras, L. Saitta (Eds.), Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, Including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004, IOS Press, 2004, pp. 388–392.
[100] D. Pearce, A. Valverde, Quantified equilibrium logic and foundations for answer set programs, in: M.G. de la Banda, E. Pontelli (Eds.), Logic Programming, 24th International Conference, Proceedings, ICLP 2008, Udine, Italy, December 9-13 2008, Springer, 2008, pp. 546–560.
[101] D. Pearce, A. Valverde, Synonymous theories and knowledge representations in answer set programming, J. Comput. Syst. Sci. 78 (2012) 86–104, https://doi.org/10.1016/j.jcss.2011.02.013.

[102] D.A. Plaisted, Theorem proving with abstraction, Artif. Intell. 16 (1981) 47–108.
[103] E. Pontelli, T.C. Son, O. Elkhatib, Justifications for logic programs under answer set semantics, Theory Pract. Log. Program. 9 (2009) 1–56.
[104] J. Pührer, H. Tompits, Casting away disjunction and negation under a generalisation of strong equivalence with projection, in: E. Erdem, F. Lin, T. Schaub (Eds.), Logic Programming and Nonmonotonic Reasoning, 10th International Conference, Proceedings, LPNMR 2009, Potsdam, Germany, September 14-18, 2009, Springer, 2009, pp. 264–276.
[105] P. Riddle, J. Douglas, M. Barley, S. Franco, Improving performance by reformulating PDDL into a bagged representation, in: Proceedings of the Workshop on Heuristics and Search for Domain-Independent Planning, HSDIP 2016, 2016, pp. 28–36.
[106] E.D. Sacerdoti, Planning in a hierarchy of abstraction spaces, Artif. Intell. 5 (1974) 115–135.
[107] Y. Sagiv, Optimizing datalog programs, in: Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM, 1987, pp. 349–362.
[108] L. Saitta, J.D. Zucker, Abstraction in Artificial Intelligence and Complex Systems, vol. 456, Springer, 2013.
[109] Z.G. Saribatur, T. Eiter, Reactive policies with planning for action languages, in: Proceedings of the 15th European Conference on Logics in Artificial Intelligence, JELIA 2016, in: Lecture Notes in Computer Science, vol. 10021, Springer, 2016, pp. 463–480.
[110] Z.G. Saribatur, T. Eiter, Omission-based abstraction for answer set programs, in: Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning, KR 2018, AAAI Press, 2018, pp. 42–51.
[111] Z.G. Saribatur, T. Eiter, Omission-based abstraction for answer set programs, Theory Pract. Log. Program. (2020) 1–51, Extended version of [110].
[112] Z.G. Saribatur, T. Eiter, A semantic perspective on omission abstraction in ASP, in: Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, IJCAI Organization, 2020, pp. 733–737.
[113] Z.G. Saribatur, P. Schüller, T. Eiter, Abstraction for non-ground answer set programs, in: Proceedings of the 16th European Conference on Logics in Artificial Intelligence, JELIA 2019, in: Lecture Notes in Computer Science, Springer, 2019, pp. 576–592.
[114] T. Schaub, S. Woltran, Answer set programming unleashed!, Kidney Int. 32 (2018) 105–108.
[115] C. Schulz, F. Toni, ABA-based answer set justification, Theory Pract. Log. Program. (2013) 4–5.
[116] J. Seipp, M. Helmert, Counterexample-guided Cartesian abstraction refinement, in: Proceedings of the 23rd International Conference on Automated Planning and Scheduling, ICAPS, 2013, 2013.
[117] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artif. Intell. 138 (2002) 181–234.
[118] S. Srivastava, N. Immerman, S. Zilberstein, A new representation and associated algorithms for generalized planning, Artif. Intell. 175 (2011) 615–647.
[119] H. Turner, Strong equivalence made easy: nested expressions and weight constraints, Theory Pract. Log. Program. 3 (2003) 609–622.
[120] A. Van Gelder, K.A. Ross, J.S. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (1991) 619–649.
[121] A. Weinzierl, Blending lazy-grounding and CDNL search for answer-set solving, in: Logic Programming and Nonmonotonic Reasoning - 14th International Conference, Proceedings, LPNMR 2017, Espoo, Finland, July 3-6, 2017, 2017, pp. 191–204.
[122] S. Woltran, A common view on strong, uniform, and other notions of equivalence in answer-set programming, Theory Pract. Log. Program. 8 (2008) 217–234, https://doi.org/10.1017/S1471068407003250.
[123] S. Zhang, F. Yang, P. Khandelwal, P. Stone, Mobile robot planning using action language BC with an abstraction hierarchy, in: Proc. International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR, Springer, 2015, pp. 502–516.