LOGCOMP

RESEARCH

REPORT

INSTITUT FÜR LOGIC AND COMPUTATION

FACHBEREICH WISSENSBASIERTE SYSTEME

OMISSION-BASED ABSTRACTION FOR ANSWER SET PROGRAMS

ZEYNEP G. SARIBATUR THOMAS EITER

LOGCOMP RESEARCH REPORT 18-06

DECEMBER 2018

Institut für Logic and Computation
FB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

LOGCOMP RESEARCH REPORT

LOGCOMP RESEARCH REPORT 18-06, DECEMBER 2018

OMISSION-BASED ABSTRACTION FOR ANSWER SET PROGRAMS

Zeynep G. Saribatur[1] Thomas Eiter[1]

Abstract. Abstraction is a well-known approach to simplify a complex problem by over-approximating it with a deliberate loss of information. It was not considered so far in Answer Set Programming (ASP), a convenient tool for problem solving. We introduce a method to automatically abstract ASP programs that preserves their structure by reducing the vocabulary while ensuring an over-approximation (i.e., each original answer set maps to some abstract answer set). This allows for generating partial answer set candidates that can help with approximation of reasoning. Computing the abstract answer sets is intuitively easier due to a smaller search space, at the cost of encountering spurious answer sets. Faithful (non-spurious) abstractions may be used to represent projected answer sets and to guide solvers in answer set construction. For dealing with spurious answer sets, we employ an ASP debugging approach to help with abstraction refinement, which determines atoms as badly omitted and adds them back in the abstraction. As a show case, we apply abstraction to explain unsatisfiability of ASP programs in terms of blocker sets, which are the sets of atoms such that abstraction to them preserves unsatisfiability. Their usefulness is demonstrated by experimental results.

[1]Institute of Logic and Computation, TU Wien; email: (eiter | zeynep)@kr.tuwien.ac.at.

Acknowledgements: This work has been supported by the Austrian Science Fund (FWF) project W1255-N23.

This article is a revised and extended version of the paper presented at the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018), October 30–November 2, 2018, Tempe, Arizona, USA.

Contents

1 Introduction

Abstraction is an approach that is widely used in Computer Science and AI in order to simplify problems, cf. (Clarke et al., 1994; Kouvaros and Lomuscio, 2015; Banihashemi et al., 2017; Giunchiglia and Walsh, 1992; Geißer et al., 2016). When computing solutions for difficult problems, abstraction allows to omit details and reduce the scenarios to ones that are easier to deal with and to understand. Such an approximation results in achieving a smaller or simpler state space, at the price of introducing spurious solutions. The well-known counterexample guided abstraction and refinement (CEGAR) approach (Clarke et al., 2003) is based on starting with an initial abstraction on a given program and checking the desired property over the abstract program. Upon encountering spurious solutions, the abstraction is refined by removing the spurious transitions observed through the solution, so that the spurious solution is eliminated from the abstraction. This iteration continues until a concrete solution is found.

Surprisingly, abstraction has not been considered much in the context of nonmonotonic knowledge representation and reasoning, and specifically not in Answer Set Programming (ASP) (Brewka et al., 2011). Simplification methods such as equivalence-based rewriting (Gebser et al., 2008; Pearce, 2004), partial evaluation (Brass and Dix, 1997; Janhunen et al., 2006), or forgetting (see (Leite, 2017) for a recent survey), have been extensively studied. However, these methods strive for preserving the semantics of a program, while abstraction may change the latter and lead to an over-approximation of the models (answer sets) of a program, in a modified language.

In this paper, we make the first step towards employing the concept of abstraction in ASP. We are focused on abstraction by omitting atoms from the program and constructing an abstract program with the smaller vocabulary, by ensuring that the original program is over-approximated, i.e., every original answer set can be mapped to some abstract answer set. Due to the decreased size of the search space, finding an answer set in the abstract program is easier, while one needs to check whether the found abstract answer set is concrete. As spurious answer sets can be introduced, one may need to go over all abstract answer sets until a concrete one is found. If the original program has no answer set, all encountered abstract answer sets will be spurious. To eliminate spurious answer sets, we use a CEGAR inspired approach, by finding a cause of the spuriousness with ASP debugging (Brain et al., 2007) and refining the abstraction by adding back some atoms that are deemed to be “badly-omitted".

An interesting application area for such an omission-based abstraction in ASP is finding an *explanation* for unsatisfiability of programs. Towards this problem, debugging inconsistent ASP programs has been investigated, for instance, in (Brain et al., 2007; Gebser et al., 2008; Oetsch et al., 2010; Dodaro et al., 2015), based on providing the reason for why an answer set expected by the user is missed. However, these methods do not address the question why the program has no answer set. We approach the unsatisfiability of an ASP program differently, with the aim to obtain a projection of the program that shows the cause of the unsatisfiability, without an initial idea on expected solutions. For example, consider the graphs shown in Figure 1. The one in Figure 1(a) is not 2-colorable due to the subgraph induced by the nodes 1-2-3, while the one in Figure 1(b) is not 3-colorable due to the subgraph of the nodes 1-2-3-4. From the original programs that encode this problem, abstracting away the rules that assigns colors to the nodes not involved in these subgraphs should still keep the unsatisfiability, thus showing the actual reason of non-colorability of the graphs. This is related to the well-known notion of minimal unsatisfiable subsets (*unsatisfiable cores*) (Liffiton and Sakallah, 2008; Lynce and Silva, 2004) that has been investigated in the ASP context (Alviano and Dodaro, 2016; Andres et al., 2012), but is less sensitive to the issue of foundedness as it arises from rule dependencies (for further discussion see Related Work).

Our contributions in this paper are briefly summarized as follows.

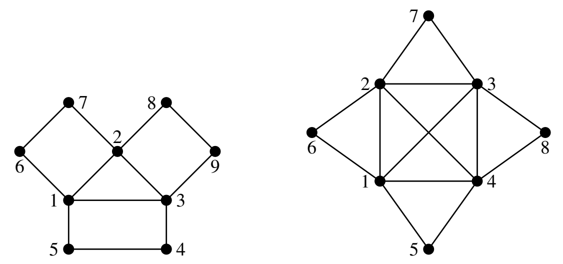

Figure 1: Graph coloring instances

(a) A non 2-colorable graph (b) A non 3-colorable graph

- We introduce a method to abstract ASP programs Π by omitting atoms in order to obtain an over-approximation of the answer sets of Π. That is, a program Π' is constructed such that each answer set I of Π is abstracted to some answer set I' of Π'. While this abstraction is many to one, *spurious* answer sets of Π' may exist that do not correspond to any answer set of Π.
- We present a refinement method inspired by ASP debugging approaches to catch the badly omitted atoms through the encountered spurious answer sets.
- We introduce the notion of *blocker sets* as sets of atoms such that abstraction to them preserves unsatisfiability of a program. A minimal program to the minimal cause of unsatisfiability.
- We derive complexity results for the notions, such as for checking for spurious answer sets, for finding minimal sets of atoms to put back in the refinement to eliminate a spurious solution, and for computing a minimal blocker for a program. In particular, we characterize the complexity of these problems in terms of suitable complexity classes, which unsurprisingly are intractable in general.
- We report about experiments focusing on unsatisfiable programs and investigate computing minimal blockers of programs. We compare the results of the abstraction and refinement approach starting with an initial abstraction (*bottom-up*) with a naive *top-down* approach that omits atoms one-by-one if their omission preserves unsatisfiability, and observe that the bottom-up approach can obtain smaller sized blockers.

Overall, abstraction by omission appears to be of interest for ASP, which besides explaining unsatisfiability can be utilized, among other applications, to over-approximate reasoning and to represent projected answer sets.

Organization The remainder of this article is organized as follows. After recalling in the next section some necessary concepts and fixing the notation, we introduce in Section 3 program abstraction by atom omission and consider some of its basic semantics properties. In Section 4 we study computational complexity issues for relevant reasoning tasks around omission, while in Section 5 we turn to the question of abstraction refinement. As an application of abstraction, we show in Section 6 how it can be used to find reasons for unsatisfiability of programs and present results obtained by an experimental prototype implementation. The subsequent Section 7 discusses some extensions and possible optimizations, while in Section 8 we address related work. The final Section 9 gives a short summary and concludes with an outlook on future research.

2 Preliminaries

We consider logic programs Π with rules r of the form

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

where each α_i is a first-order atom[1] and *not* is default negation; r is a *constraint*, if α_0 is falsity ($\bot$, then omitted) and a *fact*, if $n=0$. We also write r as $\alpha_0 \leftarrow B(r)$, where $H(r) = \alpha_0$ is the *head* of r, 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, respectively; furthermore, we let $B^\pm(r) = B^+(r) \cup B^-(r)$. We occasionally omit r from $B^\pm(r)$, $B(r)$ etc. if r is understood. As a common syntactic extension, we also consider *choice rules* of the form $\{\alpha\} \leftarrow B$, which stands for the rules $\alpha \leftarrow B, not\ \overline{\alpha}$ and $\overline{\alpha} \leftarrow B, not\, \alpha$, where $\overline{\alpha}$ is a new atom.

Semantically, Π induces a set of answer sets (Gelfond and Lifschitz, 1991), which are Herbrand models of Π that are justified by the rules. For a ground (variable-free) program Π, its answer sets are the Herbrand interpretations, i.e., subsets $I \subseteq \mathscr{A}$ of the ground atoms $\mathscr{A}$ of Π, such that I is a minimal model of $f\Pi^I = \{r \in grd(\Pi) \mid I \models B(r)\}$ (Faber et al., 2004). The answer sets of a non-ground program Π are the ones of its grounding $grd(\Pi) = \bigcup_{r\in\Pi} grd(r)$, where $grd(r)$ is the set of all instantiations of r over the Herbrand universe of Π (the set of ground terms constructible from the alphabet of Π). The set of answer sets of a program Π is denoted as $AS(\Pi)$. A program Π is *unsatisfiable*, if $AS(\Pi) = \emptyset$. Throughout this paper, unless stated otherwise we consider ground (propositional) programs, i.e., $\Pi = grd(\Pi)$ holds.

Example 1 *Consider the program* $\Pi = \{c \leftarrow not\ d.;\ d \leftarrow not\ c.;\ a \leftarrow not\ b, c.;\ b \leftarrow d.\}$ *that has two answer sets, viz.* $I_1 = \{c,a\}$ *and* $I_2 = \{d,b\}$*; indeed,* $\Pi^{I_1} = \{c \leftarrow not\ d.;\ a \leftarrow not\ b, c.\}$ *and* I_1 *is a minimal model of* Π^{I_1}*; similarly,* $\Pi^{I_2} = \{d \leftarrow not\ c.;\ b \leftarrow d.\}$ *has* I_2 *among its minimal models.*

The *dependency graph* of a program Π, denoted G_Π, has vertices $\mathscr{A}$, (positive) edges from any $\alpha_0 = H(r)$ to any $\alpha_1 \in B^+(r)$ and (negative) edges from any $\alpha_0 = H(r)$ to any $\alpha_2 \in B^-(r)$, for all $r \in \Pi$. E.g., in Example 1 G_Π has positive edges $a \rightarrow c$ and $b \rightarrow d$ and negative edges $c \rightarrow d$, $d \rightarrow c$ and $a \rightarrow b$. An *odd loop* means that an atom $\alpha \in \mathscr{A}$ depends recursively on itself through an odd number of negative edges in G_Π; constraints are viewed as simple odd loops. As well-known, Π is satisfiable, if it contains no odd loop. The program Π in Example 1, e.g., has no odd loop, and thus (as already seen) has some answer set.

To group the rules with the same head α, we use $def(\alpha,\Pi) = \{r \in \Pi \mid H(r) = \alpha\}$. An atom α is *unsupported* by an interpretation I if for each $r \in def(\alpha,\Pi)$, $B^+(r) \nsubseteq I$ or $B^-(r) \cap I \neq \emptyset$ (Van Gelder et al., 1991). A set $A \subseteq \mathscr{A}$ of atoms is *unfounded* w.r.t an interpretation I, if atoms in A only have support by themselves, i.e., a loop only with positive edges in the dependency graph.

3 Abstraction by Omission

Our aim is to over-approximate a given program through constructing a simpler program by reducing the vocabulary and ensuring that the results of reasoning on the original program are not lost, at the cost of obtaining spurious answer sets. We propose the following definition for abstraction of answer set programs.

Definition 1 *Given two programs* Π *and* Π' *with* $|\mathscr{A}| \geq |\mathscr{A}'|$*, where* $\mathscr{A}, \mathscr{A}'$ *are sets of ground atoms of* Π *and* Π'*, respectively,* Π' *is an* abstraction *of* Π *if there exists a mapping* $m : \mathscr{A} \rightarrow \mathscr{A}' \cup \{\top\}$ *such that for any answer set* I *of* Π*,* $I' = \{m(\alpha) \mid \alpha \in I\}$ *is an answer set of* Π'.

[1]Lifting the framework to programs with strong negation is easily possible, where as usual negative literals $\neg p(\vec{t})$ are viewed as atoms of a positive predicate $\neg p$ and with an additional constraint $\leftarrow p(\vec{t}), \neg p(\vec{t})$.

We refer to m as an *abstraction mapping*. This abstraction notion gives the possibility to do clustering over atoms of the program. One approach to do this is to omit some of the atoms from the program, i.e., cluster them into $\top$, and consider the abstract program which is over the remaining atoms. In this paper, we focus on such an omission-based abstraction.

Definition 2 *Given a set $A \subseteq \mathscr{A}$ of atoms, an* omission (abstraction) mapping *is $m_A : \mathscr{A} \to \mathscr{A} \cup \{\top\}$ such that $m_A(\alpha) = \top$ if $\alpha \in A$ and $m_A(\alpha) = \alpha$ otherwise.*

An omission mapping removes the set A of atoms from the vocabulary and keeps the rest. We refer to A as the *omitted atoms*.

Example 2 *Consider the below programs Π_1, Π_2 and Π_3 and let the set A of atoms to be omitted to be $\{b\}$.*

	Π_1	Π_2	Π_3
	$c \leftarrow not\ d.$	$c \leftarrow not\ d.$	$\{a\}.$
	$d \leftarrow not\ c.$	$d \leftarrow not\ c.$	$\{c\} \leftarrow a.$
	$a \leftarrow not\ b, c.$	$\{a\} \leftarrow c.$	$d \leftarrow not\ a.$
	$b \leftarrow d.$		
AS	$\{c,a\},\{d,b\}$	$\{c,a\},\{d\},\{c\}$	$\{c,a\},\{d\},\{a\}$

Observe that for $I'_1 = \{m_A(c), m_A(a)\} = \{c,a\}$ we have $I'_1 \in AS(\Pi_2)$ and $I'_1 \in AS(\Pi_3)$ and that for $I'_2 = \{m_A(d),\ m_A(b)\} = \{d\}$ we have $I'_2 \in AS(\Pi_2)$ and $I'_2 \in AS(\Pi_3)$. Thus, according to Definition 1, both of the programs Π_2 and Π_3 are an abstraction of Π_1. Moreover, they are over-approximations, as they have answer sets $\{c\}$ and $\{a\}$, respectively, which cannot be mapped back to the answer sets of Π_1.

Although both Π_2 and Π_3 are abstractions, notice that the structure of Π_2 is more similar to Π_1, while Π_3 has an entirely different structure of rules.

Next we show a systematic way of building, given an ASP program and a set A of atoms, an abstraction of Π by omitting the atoms in A that we denote by $omit(\Pi, A)$. The aim is to ensure that every original answer set of Π is mapped to some abstract answer set of $omit(\Pi, A)$, while (unavoidably) some spurious abstract answer sets may be introduced. Thus, an over-approximation of the original program Π is achieved.

3.1 Program Abstraction

The basic method is to project the rules to the non-omitted atoms and introduce choice when an atom is omitted from a rule body, in order to make sure that the behavior of the original rule is preserved.

We build from Π an abstract program $omit(\Pi, A)$ according to the abstraction m_A. For every rule $r : \alpha \leftarrow B(r)$ in Π,

$$omit(r,A) = \begin{cases} r & \text{if } A \cap B^{\pm} = \emptyset \wedge \alpha \notin A, & (a) \\ \{\alpha\} \leftarrow B^{+}(r) \setminus A, not\ (B^{-}(r) \setminus A) & \text{if } A \cap B^{\pm} \neq \emptyset \wedge \alpha \notin A \cup \{\bot\}, & (b) \\ \emptyset & \text{otherwise.} & (c) \end{cases}$$

In (a), we keep the rule as it is, if it does not contain any omitted atom. Item (b) is for the case when the rule is not a constraint and the rule head is not in A. Then the body of the rule is projected onto the remaining atoms, and a choice is introduced to the head. Note that we treat default negated atoms, $B^{-}(r)$, similarly, i.e., if some $\alpha \in B^{-}(r) \cap A$, then we omit *not* α from $B(r)$. As for the remaining cases (either the rule head is in A or the rule is a constraint), the rule is omitted by item (c). We use $\emptyset$ as a symbol for picking no rule.

We sometimes denote $omit(\Pi, A)$ as $\widehat{\Pi}_{\overline{A}}$, where $\overline{A} = \mathscr{A} \setminus A$, to emphasize that it is an abstract program constructed with the remaining atoms $\overline{A}$. For an interpretation I and a collection S of atoms, $I|_{\overline{A}}$ and $S|_{\overline{A}}$ denotes the projection to the atoms in $\overline{A}$.

Example 3 *Consider a program* Π *and its abstraction* $\widehat{\Pi}_{\overline{A}}$ *for* $A = \{b, d\}$*, according to the above steps.*

	Π	$\widehat{\Pi}_{\overline{A}}$
	$c \leftarrow not\ d.$	$\{c\}.$
	$d \leftarrow not\ c.$	
	$a \leftarrow not\ b, c.$	$\{a\} \leftarrow c.$
	$b \leftarrow d.$	
AS	$\{c,a\}, \{d,b\}$	$\{\}, \{c\}, \{c,a\}$

For $I'_1 = \{m_A(c), m_A(a)\} = \{c, a\}$ *we have* $I'_1 \in AS(\widehat{\Pi}_{\overline{A}})$ *and for* $I'_2 = \{m_A(d), m_A(b)\} = \{\}$ *we have* $I'_2 \in AS(\widehat{\Pi}_{\overline{A}})$*. Thus, every answer set of* Π *can be mapped to some answer set of* $\widehat{\Pi}_{\overline{A}}$*, when the omitted atoms are projected away, i.e.,* $AS(\Pi)|_{\overline{A}} = \{\{c,a\}, \{\}\} \subseteq \{\{c,a\}, \{\}, \{c\}\} = AS(\widehat{\Pi}_{\overline{A}})$.

Notice that in $\widehat{\Pi}_{\overline{A}}$, constraints are omitted if the body contains an omitted atom (item (c)). If instead the constraint gets shrunk by just omitting the atom from the body, then for some interpretation $\hat{I}$, the body may be satisfied, causing $\hat{I} \notin AS(\widehat{\Pi}_{\overline{A}})$, while this was not the case in Π for any $I \in AS(\Pi)$ with $I|_{\overline{A}} = \hat{I}$. Thus I cannot be mapped to an abstract answer set of $\widehat{\Pi}_{\overline{A}}$, i.e., $\widehat{\Pi}_{\overline{A}}$ is not an over-approximation of Π. The next example illustrates this.

Example 4 (Example 3 continued) *Consider an additional rule* $\{\leftarrow c, b.\}$ *in* Π*, which does not change its answer sets. If however in the abstraction* $\widehat{\Pi}_{\overline{A}}$ *this constraint only gets shrunk to* $\{\leftarrow c.\}$*, by omitting b from its body, we get* $AS(\widehat{\Pi}_{\overline{A}}) = \{\}$*. This causes to have no abstract answer set to which the original answer set* $\{c, a\}$ *can be mapped to. Omitting the constraint from* $\widehat{\Pi}_{\overline{A}}$ *as described above avoids such cases of losing the original answer sets in the abstraction.*

Abstracting choice rules We focused above on rules of the form $\alpha \leftarrow B$ only. However, the same principle is applicable to choice rules $r : \{\alpha\} \leftarrow B(r)$. When building $omit(r, A)$, item (a) keeps the rule as it is, item (b) removes the omitted atom from $B(r)$ and keeps the choice in the head, and item (c) omits the rule. This would be syntactically different from considering the expanded version (1) $\alpha \leftarrow B(r), not\ \overline{\alpha}$. (2) $\overline{\alpha} \leftarrow B(r), not\ \alpha$. where $\overline{\alpha}$ is an auxiliary atom. If α is omitted, the rule (2) turns into a guessing rule, but it is irrelevant as $\overline{\alpha}$ occurs nowhere else. If α is not omitted but some atom in B, both rules are turned into guessing rules and the same answer set combinations are achieved as with keeping r as a choice rule in item (b). However, the number of auxiliary atoms would increase, in contrast to treating choice rules r genuinely.

3.2 Over-Approximation

The following result shows that $omit(\Pi, A)$ can be seen as an over-approximation of Π.

Theorem 1 *For every answer set* $I \in AS(\Pi)$ *and atoms* $A \subseteq \mathscr{A}$*, it holds that* $I|_{\overline{A}} \in AS(omit(\Pi, A))$.

Proof. Towards a contradiction, assume I is an answer set of Π, but $I|_{\overline{A}}$ is not an answer set of $omit(\Pi, A)$. This can occur because either (i) $I|_{\overline{A}}$ is not a model of $\Pi' = omit(\Pi, A)^{I|_{\overline{A}}}$ or (ii) $I|_{\overline{A}}$ is not a minimal model of Π'.

(i) If $I|_{\overline{A}}$ is not a model of Π', then there exists some rule $r \in \Pi'$ such that $I|_{\overline{A}} \models B(r)$ and $I|_{\overline{A}} \not\models H(r)$. By the construction of $omit(\Pi, A)$, r is not obtained by case (b), i.e., by modifying some original rule to get rid of A, because then r would be a choice rule with head $H(r) = \{\alpha\}$, and r would be satisfied. Consequently, r is a rule from case (a), and thus $r \in \Pi$. We note that $I|_{\overline{A}}$ and I coincide on all atoms that occur in r. Thus, $I|_{\overline{A}} \models B(r)$ implies that $I \models B(r)$, and as $I \models r$, it follows $I \models H(r)$, which then means $I|_{\overline{A}} \models H(r)$; this is a contradiction.

(ii) Suppose $I' \subset I|_{\overline{A}}$ is a model of Π'. We claim that then $J = I' \cup (I \cap A) \subset I$ is a model of Π^I, which would contradict that $I \in AS(\Pi)$. Assume that $J \not\models \Pi^I$. Then J does not satisfy some rule $r : \alpha \leftarrow B(r)$ in Π^I, i.e., $J \models B(r)$ but $J \not\models \alpha$, i.e., $\alpha \notin J$. The rule r can either be (a) a rule which is not changed for Π', (b) a rule that was changed to $\{\alpha\} \leftarrow \widehat{B}$ in Π', or (c) a rule that was omitted, i.e., $\alpha \in A$. In each case (a)-(c), we arrive at a contradiction:

(a) Since $r \in \Pi^I$ and r involves no atom in A, we have $r \in \Pi'$. As $I|_{\overline{A}} \models r$ and $J|_{\overline{A}}$ coincides with $I|_{\overline{A}}$, we have that $J|_{\overline{A}} \models r$, and thus $J \models r$; this contradicts $J \not\models \alpha$.

(b) By definition of J, we have $\alpha \in I|_{\overline{A}} \setminus I'$. Since $J \models B(r)$, it follows that $J|_{\overline{A}} \models \widehat{B}$ and since $I' = J|_{\overline{A}}$ that $I' \models \widehat{B}$. As I' is a model of Π', we have that I' satisfies the choice atom $\{\alpha\}$ in the head of the rewritten rule, i.e., either (1) $\alpha \in I'$ or (2) $\alpha \notin I'$; but (1) contradicts $\alpha \in I|_{\overline{A}} \setminus I'$, while (2) means that I' is not a smaller model of Π' than $I|_{\overline{A}}$, as then $\alpha' \in I' \setminus I|_{\overline{A}}$ would hold, which is again a contradiction.

(c) As r is in Π^I, we have $I \models B(r)$ and since I is an answer set of Π, that $I \models \alpha$. As $\alpha \notin J$, by construction of J it follows that $\alpha \notin I$, which contradicts $I \models \alpha$. $\square$

By introducing choice rules for any rule that contains an omitted atom, all possible cases that would be achieved by having the omitted atom in the rule are covered. Thus, the abstract answer sets cover the original answer sets. On the other hand, not every abstract answer set may cover some original answer set, which motivates the following notion.

Definition 3 *Given a program Π and a set A of atoms, an answer set $\hat{I}$ of $omit(\Pi, A)$ is* concrete*, if $\hat{I} \in AS(\Pi)|_{\overline{A}}$ holds, and* spurious *otherwise.*

In other words, a spurious abstract answer set $\hat{I}$ can not be completed to any original answer set, i.e., no extension $I = \hat{I} \cup X$ of $\hat{I}$ to all atoms (where $X \subseteq A$) is an answer set of Π. This can be alternatively defined in the following way. We introduce the following set of constraints for A and $\hat{I}$:

$$Q_{\hat{I}}^{\overline{A}} = \{\bot \leftarrow not\ \alpha \mid \alpha \in \hat{I}\} \cup \{\bot \leftarrow \alpha \mid \alpha \in \overline{A} \setminus \hat{I}\}. \quad (2)$$

Informally, $Q_{\hat{I}}^{\overline{A}}$ is a query for an answer set that concides on the non-omitted atoms with $\hat{I}$. The following is then easy to see.

Proposition 2 *For any program Π and set A of atoms, an abstract answer set $\hat{I} \in AS(omit(\Pi, A))$ is spurious iff $\Pi \cup Q_{\hat{I}}^{\overline{A}}$ is unsatisfiable.*

Example 5 (Example 3 continued) *The program $\widehat{\Pi}_{\overline{A}}$ constructed for $\overline{A} = \{a, c\}$ has the answer set collection $AS(\widehat{\Pi}_{\overline{A}}) = \{\{\}, \{c\}, \{c, a\}\}$. The abstract answer sets $\hat{I}_1 = \{\}$ and $\hat{I}_2 = \{c, a\}$ are concrete since they can be extended to the answers sets $I_1 = \{d, b\}$ and $I_2 = \{c, a\}$ of Π, as $I_1|_{\overline{A}} = \hat{I}_1$ and $I_2|_{\overline{A}} = \hat{I}_2$, respectively. On the other hand, the abstract answer set $\hat{I} = \{c\}$ is spurious: the program $\Pi \cup Q_{\hat{I}}^{\overline{A}}$, where*

$Q^{\overline{A}}_{\hat{I}} = \{\bot \leftarrow not\ c.;\ \bot \leftarrow a.\}$ is unsatisfiable, since the constraints in $Q^{\overline{A}}_{\hat{I}}$ require c is true and a is false, which in turn affects that b and d must be false in Π as well; this however violates rule $a \leftarrow not\ b, c.$ in Π.

3.2.1 Refining abstractions

Upon encountering a spurious answer set, one can either continue checking other abstract answer sets until a concrete one is found, or *refine* the abstraction in order to reach an abstract program with less spurious answer sets. Formally, refinements are defined as follows.

Definition 4 *Given a omission mapping $m_A = \mathscr{A} \rightarrow \mathscr{A} \cup \{\top\}$, a mapping $m_{A'} = \mathscr{A} \rightarrow \mathscr{A} \cup \{\top\}$ is a* refinement *of m_A if $A' \subseteq A$.*

Intuitively, a refinement is made by adding some of the omitted atoms back.

Example 6 (Example 3 continued) *A mapping that omits the set $A' = \{b\}$ is a refinement of the mapping that omits $A = \{b, d\}$, as d is added back. This affects that in the abstraction program the choice rule $\{c\}$. is turned back to $c \leftarrow not\ d.$ and the rule $d \leftarrow not\ c.$ is undeleted, i.e., $omit(\Pi, A') = \{c \leftarrow not\ d.; d \leftarrow not\ c.; \{a\} \leftarrow c\}$, which has the abstract answer sets $\hat{J}_1 = \{d\}$, $\hat{J}_2 = \{c, a\}$ and $\hat{J}_3 = \{c\}$. Note that while $\hat{J}_1$ and $\hat{J}_2$ are concrete, $\hat{J}_3$ is spurious; intuitively, adding d back does not eliminate the spurious answer set $\{c\}$ of $omit(\Pi, A)$.*

The previous example motivates us to introduce a notion for sets of omitted atoms that need to be added back in order to get rid of a spurious answer set.

Definition 5 *Let $\hat{I} \in AS(omit(\Pi, A))$ be any spurious abstract answer set of a program Π for omitted atoms A. A* put-back set *for $\hat{I}$ is any set $PB \subseteq A$ of atoms such that no abstract answer set $\hat{J}$ of $omit(\Pi, A')$ where $A' = A \setminus PB$ exists with $\hat{J}|_{\overline{A}} = \hat{I}$.*

That is, re-introducing the put-back atoms in the abstraction, the spurious answer set I is *eliminated* in the modified abstract program. Notice that multiple put-back sets (even incomparable ones) are possible, and the existence of some put-back set is guaranteed, as putting all atoms back, i.e., setting $PB = A$, eliminates the spurious answer set.

Example 7 (Example 3 continued) *The discussion in Example 6 shows that $\{d\}$ is not a put-back set, for the spurious answer set $\hat{I} = \{c\} \in \widehat{\Pi}_{\overline{A}}$, and neither $\{b\}$ is a put-back set: the abstract program for $A' = A \setminus \{b\} = \{d\}$ is $omit(\Pi, A') = \{\{c\}.; a \leftarrow not\, b, c.; \{b\}.\}$, which has $\hat{I}$ among its abstract answer sets. Thus, $\hat{I}$ has only the trivial put-back set $\{b, d\}$.*

In practice, small put-back sets are intuitively preferable to large ones as they keep higher abstraction; we shall consider such preference in Section 4.

3.3 Properties of Omission Abstraction

We now consider some basic but useful semantic properties of our formulation of program abstraction. Notably, it amounts to the original program in the extremal case and reflects the inconsistency of it in properties of spurious answer sets.

Proposition 3 *For any program Π,* *(i)* $omit(\Pi, \emptyset) = \Pi$ *and* $omit(\Pi, \mathscr{A}) = \emptyset$.

(ii) $AS(\Pi) = \emptyset$ *iff* $I = \{\}$ *is spurious w.r.t.* $A = \mathscr{A}$.

(iii) $AS(omit(\Pi, A)) = \emptyset$ *implies* $AS(\Pi) = \emptyset$.

(iv) $AS(\Pi) = \emptyset$ *iff some* $A \subseteq \mathscr{A}$, *has only spurious answer sets iff every* $omit(\Pi, A)$, $A \subseteq \mathscr{A}$, *has only spurious answer sets.*

Proof.

(i) Omitting the set $\emptyset$ from Π causes no change in the rules, while omitting the set $\mathscr{A}$ causes all the rules to be omitted.

(ii) Since $\hat{I} = \{\}$ and $A = \mathscr{A}$, we have $Q_{\hat{I}}^{\overline{A}} = \{\}$. Thus, by the alternative definition, $I = \{\}$ is spurious w.r.t. $A = \mathscr{A}$ iff $AS(\Pi \cup Q_{\hat{I}}^{\overline{A}}) = \emptyset$ iff $AS(\Pi) = \emptyset$.

(iii) Corollary of Theorem 1.

(iv) If $AS(\Pi) = \emptyset$, then no $\hat{I} \in AS(omit(\Pi, A))$ for any $A \subseteq \mathscr{A}$ can be extended to an answer set of Π; thus, all abstract answer sets of $omit(\Pi, A)$ are spurious. This in turn trivially implies that $omit(\Pi, A)$ has for some $A \subseteq \mathscr{A}$ only spurious answer sets. Finally, assume the latter holds but $AS(\Pi) \neq \emptyset$; then Π has some answer set I, and by Theorem 1, $I|_{\overline{A}} \in AS(omit(\Pi, A))$, which would contradict that $omit(\Pi, A)$ has only spurious answer sets. □

The abstract program is built by a syntactic transformation, given the set A of atoms to be omitted. It turns out that we can omit the atoms sequentially, and the order does not matter.

Lemma 4 *For any program* Π *and for any atoms* $a_1, a_2 \in \mathscr{A}$, *it holds that* $omit(omit(\Pi, \{a_1\}), \{a_2\}) = omit(omit(\Pi, \{a_2\}), \{a_1\})$.

Proof. The rules of Π that do not contain a_1 or a_2 remain unchanged, and the rules that contain one of a_1 or a_2 will be updated at the respective abstraction steps. The rules that contain both a_1 and a_2 are treated as follows:

- Consider a rule $a_1 \leftarrow B$ with $a_2 \in B$ (wlog). Omitting first a_2 from the rule causes to have $\{a_1\} \leftarrow B \setminus \{a_2\}$, and omitting then a_1 results in omission of the rule. Omitting first a_1 from the rule causes the omission of the rule at the first abstraction step.
- Consider a rule $\alpha \leftarrow B$, with $a_1, a_2 \in B$ and $\alpha \neq a_1, a_2$. Omitting first a_2 from the rule causes to have $\{a\} \leftarrow B \setminus \{a_2\}$, and omitting then a_1 causes to have $\{a\} \leftarrow B \setminus \{a_1, a_2\}$. The same rule is obtained when omitting first a_1 and then a_2. □

An easy induction argument shows then property mentioned above.

Proposition 5 *For any program* Π *and set* $A = \{a_1, \ldots, a_n\}$ *of atoms,*

$$omit(\Pi, A) = omit(omit(\cdots(omit(\Pi, \{a_{\pi(1)}\}), \cdots \{a_{\pi(n-1)}\}), \{a_{\pi(n)}\})$$

where π *is any permutation of* $\{1, \ldots, n\}$.

Thus, the abstraction can be done one atom at a time.

Omitting atoms in a program means projecting them away from the answer sets. Thus, for a mapping m_A, the concrete answer sets in $omit(\Pi, A)$ always have corresponding answer sets in the programs computed for refinements of m_A.

Proposition 6 *Suppose $\hat{I}$ is a concrete answer set of $omit(\Pi, A)$ for a program Π and a set A of atoms. Then for every $A' \subseteq A$ some answer set $\hat{I}' \in AS(omit(\Pi, A'))$ exists such that $\hat{I}'|_{\overline{A}} = \hat{I}$.*

Proof. By Definition 3, $\hat{I} \in AS(\Pi)|_{\overline{A}}$, i.e. there exists some $I \in AS(\Pi)$ s.t. $I|_{\overline{A}} = \hat{I}$. By Theorem 1, for every $B \subseteq \mathscr{A}$, $I|_{\overline{B}} \in AS(omit(\Pi, B))$ holds, and in particular for $B \subseteq A$; we thus obtain $(I|_{\overline{B}})|_{\overline{A}} = I|_{\overline{A}} = \hat{I}$. □

The next property is convexity of spurious answer sets.

Proposition 7 *Suppose $\hat{I} \in AS(omit(\Pi, A))$ is spurious and that $omit(\Pi, A')$, where $A' \subseteq A$, has some answer set $\hat{I}'$ such that $\hat{I}'|_{\overline{A}} = \hat{I}$. Then for every A'' such that $A' \subseteq A'' \subseteq A$, it holds that $\hat{I}'|_{\overline{A''}} \in AS(omit(\Pi, A''))$ and is spurious.*

Proof. We first note that $\hat{I}'$ is spurious as well: if not, some $I \in AS(\Pi)$ exists such that $I|_{\overline{A'}} = \hat{I}'$; but then $I|_{\overline{A}} = (I|_{\overline{A'}})|_{\overline{A}} = \hat{I}'|_{\overline{A}} = \hat{I}$, which contradicts that $\hat{I}$ is spurious. Applying Theorem 1 to $omit(\Pi, A')$ and A'', we obtain that $\widehat{I'}|_{\overline{A''}}$ is an answer set of $omit(omit(\Pi, A'), A'')$, which by Proposition 5 coincides with $omit(\Pi, A'')$. Moreover, $\hat{I}'|_{\overline{A''}}$ is spurious, since otherwise $\hat{I}'$ would not be spurious either, which would be a contradiction. □

The next proposition intuitively shows that once a spurious answer set is eliminated by adding back some of the omitted atoms, no extension of this answer set will show up when further omitted atoms are added back.

Proposition 8 *Suppose that $\hat{I} \in AS(omit(\Pi, A))$ is a spurious answer set and $PB \subseteq A$ is a put-back set for $\hat{I}$. Then for every $A' \subseteq A \setminus PB$ and answer set $\hat{I}' \in AS(omit(\Pi, A \setminus (PB \cup A')))$ it holds that $\hat{I}'|_{\overline{A}} \neq \hat{I}$.*

Proof. Towards a contradiction, assume that for some $A' \subseteq A \setminus PB$ and answer set $\hat{I}' \in AS(omit(\Pi, A \setminus (PB \cup A'))$ it holds that $\hat{I}'|_{\overline{A}} = \hat{I}$. By Proposition 7, we obtain that $\hat{I}'$ is spurious and moreover that $\hat{I}'|_{\overline{A \setminus PB}} \in AS(omit(\Pi, A \setminus PB)$ and is spurious. However, as $\hat{I}'|_{\overline{A \setminus PB}} = \hat{I}|_{\overline{A \cup PB}}$, this contradicts that PB is a put-back set for $\hat{I}$. □

3.4 Faithful Abstractions

Ideally, abstraction simplifies a program but does not change its semantics. Our next notion serves to describe such abstractions.

Definition 6 *An abstraction $omit(\Pi, A)$ is* faithful*, if it has no spurious answer sets.*

Faithful abstractions are a syntactic representation of projected answer sets, i.e., $AS(omit(\Pi, A)) = AS(\Pi)|_{\overline{A}}$. They fully preserve the information contained in the answer sets, and allow for reasoning (both brave and cautious) that is sound and complete over the projected answer sets.

Example 8 (Example 3 continued) *Consider omitting the set $A = \{a, c\}$ from Π. The resulting $\widehat{\Pi}_{\overline{A}}$ is faithful, since its answer sets $\{\{\}, \{b, d\}\}$ are the ones obtained from projecting $\{a, c\}$ away from $AS(\Pi)$.*

	Π	$\widehat{\Pi}_{\overline{A}}$
	$c \leftarrow not\ d.$	
	$d \leftarrow not\ c.$	$\{d\}.$
	$a \leftarrow not\ b, c.$	
	$b \leftarrow d.$	$b \leftarrow d.$
AS	$\{c,a\},\{d,b\}$	$\{\},\{d,b\}$

However, while an abstraction may be faithful, by adding back omitted atoms the faithfulness might get lost. In particular, if the program Π is satisfiable, then $A = \mathscr{A}$ is a faithful abstraction; by adding back atoms, spurious answer sets might arise. This motivates the following notion.

Definition 7 *A faithful abstraction $omit(\Pi,A)$ is* refinement-safe*, if for all $A' \subseteq A$, $omit(\Pi,A')$ has no spurious answer sets.*

In a sense, a refinement-safe abstraction allows us to zoom in details without losing already established relationships between atoms, as they appear in the abstract answer sets, and no spuriousness check is needed. In particular, this applies to programs that are unsatisfiable. By Proposition 3-(iii), unsatisfiability of an abstraction $omit(\Pi,A)$ implies that the original program is unsatisfiable, and hence the abstraction is faithful. Moreover, we obtain:

Proposition 9 *Given Π and A, if $omit(\Pi,A)$ is unsatisfiable, then it is refinement-safe faithful.*

Proof. Assume that A is refined to some $A' \subset A$, where some atoms are added back in the abstraction, and the constructed $omit(\Pi,A')$ is not unsatisfiable, i.e., $AS(omit(\Pi,A')) \neq \emptyset$. By Theorem 1, it must hold that $AS(omit(\Pi,A')) \subseteq AS(omit(\Pi,A))$, which contradicts to the fact that $omit(\Pi,A)$ is unsatisfiable. $\square$

4 Computational Complexity

In this section, we turn to the computational complexity of reasoning tasks that are associated with program abstraction. We start with noting that constructing the abstract program and model checking on it is tractable.

Lemma 10 *Given Π and A, (i) the program $omit(\Pi,A)$ is constructible in logarithmic space, and (ii) checking whether $I \in AS(omit(\Pi,A))$ holds for a given I is feasible in polynomial time.*

As for item (i), the abstract program $omit(\Pi,A)$ is easily constructed in a linear scan of the rules in Π; item (ii) reduces then to answer set checking of an ordinary normal logic program, which is well-known to be feasible in polynomial time (and in fact **P**-complete).

However, tractability of abstract answer set checking is lost if we ask in addition for concreteness or spuriousness.

Proposition 11 *Given a program Π, a set of atoms A, and an interpretation I, deciding whether $I|_{\overline{A}}$, is a concrete (resp., spurious) abstract answer set of $omit(\Pi,A)$ is* **NP***-complete (resp.* co**NP***-complete).*

Proof. Indeed, we can guess an interpretation J of Π such that (a) $J_{\overline{A}} = I_{\overline{A}}$, (b) $J_{\overline{A}} \in AS(omit(\Pi,A))$, and (c) $J \in AS(\Pi)$. By Lemma 10, (b) and (c) are feasible in polynomial time, and thus deciding whether $I_{\overline{A}}$ is a concrete abstract answer set is in **NP**. Similarly, $I_{\overline{A}}$ is not a spurious abstract answer set iff for some J condition (a) holds and either (b) fails or (c) holds; this implies co**NP** membership.

The **NP**-hardness (resp. co**NP**-hardness) is immediate from Proposition 3 and the **NP**-completeness of deciding answer set existence. □

Thus, determining whether a particular abstract answer set causes a loss of information is intractable in general. If we do not have a candidate answer set at hand, but want to know whether the abstraction causes a loss of information with respect to all answer sets of the original program, then the complexity increases.

Theorem 12 *Given a program* Π *and a set* A *of atoms, deciding whether some* $\hat{I} \in AS(omit(\Pi, A))$ *exists that is spurious is* Σ_2^p*-complete.*

Proof. As for membership in Σ_2^p, some answer set $\hat{I} \in omit(\Pi, A)$ that is spurious can be guessed and checked by Proposition 11 with a co**NP** oracle in polynomial time. The Σ_2^p-hardness is shown by a reduction from evaluating a QBF $\exists X \forall Y\, E(X,Y)$, where $E(X,Y) = \bigvee_{i=1}^{k} D_i$ is a DNF of conjunctions $D_i = l_{i_1} \wedge \cdots \wedge l_{i_{n_i}}$ over atoms $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_m\}$ where without loss of generality in each D_i some atom from Y occurs.

We construct a program Π as follows;

$$x_i \leftarrow not\ \overline{x_i}. \tag{3}$$

$$\overline{x_i} \leftarrow not\ x_i. \quad \text{for all } x_i \in X \tag{4}$$

$$y_j \leftarrow not\ \overline{y_j}, not\ sat. \tag{5}$$

$$\overline{y_j} \leftarrow not\ y_j, not\ sat. \quad \text{for all } y_j \in Y \tag{6}$$

$$sat \leftarrow l_{i_1}^*, \dots l_{i_{n_i}}^*. \tag{7}$$

where $\overline{X} = \{\overline{x}_1, \dots \overline{x_n}\}$ and $\overline{Y} = \{\overline{y}_1, \dots \overline{y_m}\}$ are sets of fresh atoms and for each atom $a \in X \cup Y$, we let $a* = a$ and $(\neg a)^* = \overline{a}$. Furthermore, we set $A = Y \cup \overline{Y} \cup \{sat\}$.

Intuitively, the answer sets $\hat{I}$ of $omit(\Pi, A)$, which consists of all rules (3)-(4), correspond 1-1 to the truth assignments σ of X. A particular such $\hat{I} = \hat{I}_\sigma = \{x_i \in X \mid \sigma(x_i) = true\} \cup \{\overline{x_i} \mid x_i \in X, \sigma(x_i) = false\}$ is spurious, iff it can not be extended after putting back all omitted atoms to an answer set J of Π. Any such J must not include sat, as otherwise the rules (5) and (6) would not be applicable w.r.t. J, which means that all y_j and $\overline{Y_j}$ would be false in J; but then sat could not be derived from Π and J, as no rule (7) is applicable w.r.t. J by the assumption on the D_i.

Now of $\hat{I}_\sigma$ is not spurious, then some answer set J of Π as described exists. As $sat \notin J$, the rules (5) and (6) imply that exactly one of y_j and $\overline{y}_j$ is in J, for each y_j, and thus J induces an assignment μ to Y. As no rule (7) is applicable w.r.t. J, it follows that $E(\sigma(X), \mu(Y))$ evaluates to false, and thus $\forall Y E(\sigma(X), Y)$ does not evaluate to true. Conversely, if $\forall Y E(\sigma(X), Y)$ does not evaluate to true, then some answer set J of Π that coincides with $\hat{I}\_\sigma$ on $X \cup \overline{X}$ exists, and hence $\hat{I}\_\sigma$ is not spurious. In conclusion, it follows that $omit(\Pi, A)$ has some spurious answer set iff $\exists X \forall Y E(X,Y)$ evaluates to true. □

An immediate consequence of the previous theorem is that checking whether an abstraction $omit(\Pi, A)$ is faithful has complementary complexity.

Corollary 13 *Given a program* Π *and a set* $A \subseteq \mathscr{A}$ *of atoms, deciding whether* $omit(\Pi, A)$ *is faithful is* Π_2^p*-complete.*

We next consider the computation of put-back sets, which is needed for the elimination of spurious answer sets. To describe the complexity, we use some complexity classes for search problems, which generalize decision problems in that for a given input, some (possibly different or none) output values (or solutions) might be computed. Specifically, $\mathbf{FP^{NP}}$ consists of the search problems for which a solution can be computed in polynomial time with an **NP** oracle, and $\mathbf{FP^{NP}_{\|}}$ is analogous but under the restriction that all oracle calls have to be made at once in parallel. The class $\mathbf{FP}^{\Sigma^p_k}[log, wit]$, for $k \geq 1$, contains all search problems that can be solved in polynomial time with a witness oracle for Σ^p_k (Buss et al., 1993); a *witness* oracle for Σ^p_k returns in case of a yes-answer to an instance a polynomial size witness string that can be checked with an Σ^p_{k-1} oracle in polynomial time. In particular, for $k = 1$, i.e., for $\mathbf{FP^{NP}}[log, wit]$, one can use a SAT oracle and the witness is a satisfying assignment to a given SAT instance, cf. (Janota and Marques-Silva, 2016).

While an arbitrary put-back set $PB \subseteq A$ is can be trivially obtained (just set $PB = A$), computing a minimal put-back set is more involved. Specifically, we have:

Theorem 14 *Given a program* Π*, a set of atoms* A*, and a spurious answer set* $\hat{I}$ *of* $omit(\Pi,A)$*, computing (i) some* $\subseteq$*-minimal put-back set* PB *resp. (ii) some smallest size put-back set* PB *for* $\hat{I}$ *is in case (i) feasible in* $\mathbf{FP^{NP}}$ *and* $\mathbf{FP^{NP}_{\|}}$*-hard resp. is in case (ii)* $\mathbf{FP}^{\Sigma^p_2}[log, wit]$*-complete.*

Note that few $\mathbf{FP}^{\Sigma^p_2}[log, wit]$-complete problems are known. The notions of hardness and completeness are here with respect to a natural polynomial-time reduction between two problems P_1 and P_2: there are polynomial-time functions f_1 and f_2 such that (i) for every instance x_1 of P_1, $x_2 = f_1(x_1)$ is an instance of P_2, such that x_2 has solutions iff x_1 has, and (ii) from every solution s_1 of x_2, some solution $s_1 = f_2(x_1, s_2)$ is obtainable.

Proof. [Proof of Theorem 14] As for membership in (i), we can compute such a set PB by an elimination procedure as follows. Starting with $A' = \emptyset$, we repeatedly pick some atom $\alpha \in A \setminus A'$ and test the following condition:

(+) for $A'' = A' \cup \{\alpha\}$, the program $omit(\Pi,A'')$ has no answer set $\widehat{I''}$ such that $\widehat{I''}|_{\overline{A}} = \hat{I}$.

If (+) holds, we set $A' := A''$ and make the next pick from $A \setminus A'$. Upon termination, $PB = A \setminus A'$ is a minimal put-back set. The correctness of this procedure follows from Proposition 8, by which the elimination of spurious answer sets is anti-monotonic in the set A of atoms to omit. As for the effort, the test (+) can be done in polynomial time with an **NP** oracle; from this, membership in in $\mathbf{FP^{NP}}$ follows.

The hardness for $\mathbf{FP^{NP}_{\|}}$ is shown by a reduction from computing, given normal logic programs $\Pi_1, \ldots, \Pi_n$ on disjoint sets $X_1, \ldots, X_n$ of atoms, the answers $q_1, \ldots, q_n$ to whether P_i has some answer set ($q_i = 1$) or not ($q_i = 0$).

To this end, we use fresh atoms a_i, b_i, c_i and construct

$$\begin{aligned} \Pi'_i \;=\; & \{H(r) \leftarrow B(r), not\ b_i, not\ a_i \mid r \in \Pi_i\} \cup \\ & \{a_i \leftarrow x, not\ x \mid x \in X_i\} \cup \{b_i \leftarrow not\ c_i;\;\; c_i \leftarrow not\ b_i\}. \end{aligned}$$

The program Π'_i has the answer sets $\{b_i\}$ and $\{c_i\} \cup I_i$ where I_i is any answer set of Π_i. Then for $A_i = \mathscr{A} \setminus \{a_i, b_i\}$, we have that $I_i = \{a_i\}$ is a spurious answer set of $omit(\Pi'_i, A_i)$. To eliminate I_i with $A'_i \subseteq A_i$, must put all atoms $x \in X_i$ back: otherwise $omit(\Pi'_i, A'_i)$ contains $\{a_i\}$, and thus regardless of whether $c_i \in A'_i$, $omit(\Pi'_i, A'_i)$ has some answer set $\widehat{J_i}$ such that $\widehat{J_i}|_{\{a_i,b_i\}} = \hat{I}_i$. Moreover, if all X_i are put back (i.e., $A'_i = \mathscr{A} \setminus (X \cup \{a_i, b_i\}) = \{c_i\}$), then $omit(\Pi'_i, A'_i)$ has some answer set $\widehat{J_i}$ such that $\widehat{J_i}|_{\{a_i,b_i\}} = \hat{I}_i$ iff Π_i has some answer set I: if such a $\widehat{J_i}$ exists and since $\widehat{J_i} \models H(r) \leftarrow B(r), not\ b_i, not\ a_i$ for each $r \in Pi_i$, it follows that $\widehat{J_i} \models H(r) \leftarrow B(r)$ and in fact that $\widehat{J_i}$ is an answer set of $omit(\Pi'_i, A'_i)$ such that $c_i \in \widehat{J_i}$, and thus Π_i has some answer set; $I_i \in AS(\Pi_i)$ implies that $\widehat{J_i} = I_i \cup \{c\} \in AS(omit(\Pi'_i, A'_i))$. That is, $PB_i = X_i$ is a put-back set for $\widehat{I_i}$, and moreover the unique $\subseteq$-minimal put-back set iff P_i has some answer set.

$$
\begin{array}{lrr}
x_i. \quad \overline{x_i}. & i = 1\ldots,n & (9)\\
sat \leftarrow x_i, not\ x_i, \overline{x_i}, not\ \overline{x_i}. & i = 1\ldots,n & (10)\\
z_i \leftarrow not\ \overline{z_i}, x_i, not\ \overline{x_i}. & i = 1\ldots,n & (11)\\
\overline{z_i} \leftarrow not\ z_i, \overline{x_i}, not\ x_i. & i = 1\ldots,n & (12)\\
y_j \leftarrow not\ \overline{y_j}, not\ sat. & j = 1,\ldots,m & (13)\\
\overline{y_j} \leftarrow not\ y_j, not\ sat. & j = 1,\ldots,m & (14)\\
sat \leftarrow l^\circ_{i_1},\ldots l^\circ_{i_{n_i}}. & i = 1,\ldots,k & (15)\\
sat \leftarrow y_j, not\ y_j. & j = 1,\ldots,m & (16)\\
sat \leftarrow \overline{y_j}, not\ \overline{y_j}. & j = 1,\ldots,m & (17)\\
sat \leftarrow z_i, not\ z_i. & i = 1\ldots,n & (18)\\
sat \leftarrow \overline{z_i}, not\ \overline{z_i}. & i = 1\ldots,n & (19)
\end{array}
$$

Figure 2: Program rules for the proof of Theorem 14-(ii), first part

We construct the final program as $\Pi' = \bigcup_{i=1}^{n} Pi'_i \cup \{a_i \leftarrow a_j \mid 1 \leq i \neq j \leq n\}$. Then, $\hat{I} = \{a_1,\ldots,a_n\}$ is a spurious answer set of $omit(\Pi', \mathscr{A} \setminus \bigcup_{i=1}^{n}\{a_i,b_i\})$, and has a unique $\subseteq$-minimal put-back set PB such that $c_i \notin BP$ iff P_i has some answer set; this proves $\mathbf{FP}_{\parallel}^{\mathbf{NP}}$-hardness.

As for (ii), the membership in $\mathbf{FP}^{\Sigma_2^P}[log, wit]$ holds as we can decide the problem by a binary search for a put-back set of bounded size using a Σ_2^P witness oracle, where the finally obtained put-back set is output.

The $\mathbf{FP}^{\Sigma_2^P}[log, wit]$ hardness is shown by a reduction from the following problem. Given a QBF $\Phi = \exists X \forall Y\, E(X,Y)$, compute a smallest size truth assignment σ to X such that $\forall Y\, E(\sigma(X),Y)$ evaluates to true, knowing that some σ with this property exists, where the size of σ is the number of atoms set to true.

More specifically, we assume similar as in the proof of Theorem 12 that $E(X,Y) = \bigvee_{i=1}^{k} D_i$ is a DNF where every $D_i = l_{i_1} \wedge \cdots \wedge l_{i_{n_i}}$ is a conjunction of literals over $X = \{x_1,\ldots,x_n\}$ and $Y = \{y_1,\ldots,y_m\}$ that contains some literal over Y; moreover, we assume that $E(X,Y)$ is a tautology if all literals over X are removed from it. To verify the latter assumption, we may rewrite Φ to

$$\exists X \forall Y \bigvee_{x_i \in X} (x_i \wedge \neg x_i \wedge y_j) \vee (x_i \wedge \neg x_i \wedge \neg y_j) \vee E(X,Y), \tag{8}$$

for an arbitrary $y_j \in Y$, which has the desired property.

We set up a program Π with rules shown in Figure 2, where $\overline{X} = \{\bar{x}_i \mid x_i \in X\}$, $Z = \{z_1,\ldots,z_n\}$ and $\overline{Z} = \{\bar{z}_i \mid z_i \in Z\}$ are copies of X and $\overline{Y} = \{\bar{y}_j \mid y_j \in Y\}$ is a copy of Y, and l° maps a literal l over $X \cup Y$ to default literals over $Y \cup \overline{Y} \cup Z \cup \overline{Z}$ as follows:

$$l^\circ = \begin{cases} not\ z_i, & \text{if } l = \neg x_i,\\ not\ \overline{z_i}, & \text{if } l = x_i,\\ y_j, & \text{if } l = y_j,\\ \bar{y}_j & \text{if } l = \neg y_j. \end{cases}$$

We note that Π has no answer set: due to the facts x_i and $\bar{x}_i$, none of the rules (10)–(12) is applicable and $z_i, \bar{z}_i$ must be false in every answer set of Π. This in turn implies that in (15) all $not\ z_i$, $not\ \bar{z}_i$ literals are true. Now if we assume that *sat* would be true in an answer set of Π, then no rule in (13) or (14) would be applicable to derive y_j resp. $\bar{y}_j$, and then by the assumption on $E(X,Y)$ no rule (15) is applicable; this means that *sat* is not reproducible then and this not on the answer set, which is a contradiction. If on the other hand *sat* would be false in an answer set, then the rules (13) and (14) would guess a truth assignment to Y; by the tautology assumption on $E(X,Y)$, some rule (15) is applicable and derives that *sat* is true, which is again a contradiction.

We then set $A = \mathscr{A}$ and $\hat{I} = \emptyset$; clearly $\hat{I}$ is a spurious answer set of $omit(\Pi, A) = \emptyset$.

The idea behind this construction is as follows. As long as we do not put back *sat*, the abstraction program $omit(\Pi, A')$ will have some answer set. Furthermore, if we do not put back (a) either x_i or $\overline{x_i}$, for all $i = 1, \dots, n$, (b) both z_i and $\overline{z_i}$ for all $i = 1, \dots, n$ and(c) all y_j, $\overline{y_j}$, for $j = 1, \dots, m$, then we can guess by (10) resp. (16)–(19) that *sat* is true, which again means that some answer set exists. The rules (11)–(12) serve then to provide with z_i and $\overline{z_i}$ access to x_i and its negation $\neg x_i$, respectively. More in detail, if we put back x_i but not $\overline{x_i}$, then $omit(\Pi, A')$ contains the guessing rules $r_i : \{z_i\} \leftarrow not\ \overline{z_i}, x_i$ and $\overline{r_i} : \{\overline{z_i}\} \leftarrow not\ z_i, not\ x_i$ resulting from (11) and (12), respectively. As in $omit(\Pi, A')$ the fact x_i. occurs and no rule has $\overline{x_i}$ in the head, the rule $\overline{r_i}$ is inapplicable and $\overline{z_i}$ thus false; hence the rule r_i amounts to a guess $\{z_i\}$. If z_i is guessed to be true, then *not* z_i and *not* $\overline{z_i}$ faithfully represent the value of the literals $\neg x_i$ and x_i (where x_i is true); this is injected into the rules (15). On the other hand, if z_i is guessed false, then both *not* z_i and *not* $\overline{z_i}$ are true, which represents that both $\neg x_i$ and x_i are true; if on such a guess, none of the rules (15) fires (which would be necessary to have an answer set), the same holds if z_i is guessed false, as z_i and $\overline{z_i}$ occur there only negated. The case of putting back $\overline{x_i}$ but not x_i is symmetric.

The rules (13)–(14) serve to guess an assignment μ to Y (but this only works if *sat* is false). The rules (15) check whether upon a combined assignment $\sigma \cup \mu$, the formula $E(\sigma(X), \mu(Y))$ evaluates to true; if this is the case, *sat* is concluded which then however blocks the guessing in (13)–(14), and thus no answer set exists. Consequently, $E(\sigma(X), \mu(Y))$ evaluates to true for all assignments $\mu(Y)$, i.e., $\forall Y E(\sigma(X), Y)$ is true iff *sat* can be concluded for each guess on y_i and $\overline{y_i}$, i.e., no answer set is possible for it.

In conclusions, it holds that some put-back set of size $s = |X| + 2|X| + 2|Y| + 1$, which is the smallest possible here, exists iff Φ evaluates to true. Note, that if we put back a single further atom, for some $x_i \in X$ we have that $\overline{x_i}$ is also a fact in $omit(\Pi, A')$, and thus by the special form of $E(X,Y)$ in (8), regardless of how one guesses on y_j and $\overline{y_i}$, one can derive *sat* again. Thus the closest put-back set has either size s or $s+1$.

In order to discriminate among different $\sigma(X)$ and select the smallest, we add further rules:

$$sat \leftarrow not\ \overline{z_i}, c_i \tag{20}$$

$$sat \leftarrow not\ \overline{z_i}, not\ z_i, d_1, \dots, d_l \tag{21}$$

where all c_i and d_j are fresh atoms. Intuitively, when x_i is put back, then $\neg z_i$ evaluates to true and c_i must be put back as well in order to avoid guessing on *sat*. Furthermore, if both x_i and $\overline{x_i}$ are put back, which means that *not* z_i and *not* $\overline{z_i}$ are true in every answer set, then all $d_1, \dots, d_n$ must be put back as well. If exactly one of x_i and $\overline{x_i}$, for all $i = 1, \dots, n$ is put back and the corresponding assignment $\sigma(X)$ makes $\forall Y E(\sigma(X), Y)$ true, then the closest put-back set has size $s + 1 + |\sigma|$; if we let l be large enough, then putting both x_i and $\overline{x_i}$ back is more expensive than putting back a proper assignment and the associated c_i atoms. As the final program Π is constructible in polynomial time from Φ, and the desired smallest $\sigma(X)$ is easily obtained from any smallest put-back set *PB* for $\hat{I}$ the claimed result follows. □

We remark that the problem is solvable in polynomial time, if the smallest put-back set *PB* has a size bounded by a constant k. Indeed, in this case we can explore all *PB* of that size, and find all answer sets $\widehat{I'}$ of $omit(\Pi, \overline{A \cup S})$ that coincide with I on $\overline{L}$ in polynomial time.

We finally consider the problem of computing some refinement-safe abstraction that does not remove a given set A_0 of atoms.

Theorem 15 *Given a set $A_0 \subseteq \mathscr{A}$, computing (i) some $\subseteq$-maximal set $A \subseteq \mathscr{A} \setminus A_0$ resp. (ii) some $A \subseteq \mathscr{A} \setminus A_0$ of largest size such that $omit(\Pi, A)$ is a refinement-safe faithful abstraction is in case (i) in* $\mathbf{FP}^{\mathbf{NP}}$ *and* $\mathbf{FP}^{\mathbf{NP}}_{\parallel}$*-hard and in case (ii)* $\mathbf{FP}^{\Sigma^p_2}[log, wit]$*-complete, with* $\mathbf{FP}^{\Sigma^p_2}[log, wit]$*-hardness even if $A_0 = \emptyset$.*

Proof. (i) One sets $A := \emptyset$ and $S := \mathscr{A} \setminus A_0$ initially and then picks an atom α from S and sets $S := S \setminus \{\alpha\}$. One tests whether (*) omitting $A' \cup \{\alpha\}$, for every subset $A' \subseteq A$, is a faithful abstraction; if so, then one sets $A := A \cup \{\alpha\}$. Then a next atom α is picked from S etc. When this process terminates, we have a largest set A such that omitting A from Π is a faithful abstraction. Indeed, by construction the final set A fulfills that for each $A' \subseteq A$, $omit(\Pi, A')$ is faithful, and thus A is refinement-safe; furthermore A is maximal: if a larger set $A' \supset A$ would exist, then at the point when $\alpha \in A' \setminus A$ was considered in constructing A the test (*) would not have failed and $\alpha \in A$ would hold.

Notably, (*) can be tested with an **NP** oracle: the conditions fails iff for some A', the program $omit(\Pi, A' \cup \alpha)$ has a spurious answer set $\hat{I}$. In principle, the spurious check for $\hat{I}$ is difficult (a co**NP**-complete problem, by our results), but we can take advantage of knowing that $omit(\Pi, A')$ is faithful: so we only need to check whether an extension of $\hat{I}$ is an answer set of $omit(\Pi, A')$, and not of Π itself; i.e., we only need to check $\hat{I} \notin AS(omit(\Pi, A'))$ and $\hat{I} \cup \{\alpha\} \notin AS(omit(\Pi, A'))$.

(ii) The proof of $\mathbf{FP}^{\Sigma^p_2}[log, wit]$-completeness is similar as above for Theorem 14. First, we note that to decide whether some refinement-safe faithful $A \subseteq \mathscr{A} \setminus A_0$ of size $|A| \geq k$ exists is in Σ^p_2: a nondeterministic variant of the algorithm for item (i), that picks α always nondeterministically and finally checks that $|A| \geq k$ holds establishes this. We then can run a binary search, using a Σ^p_2 witness oracle, to find a refinement-safe faithful abstraction A of largest size. This shows $\mathbf{FP}^{\Sigma^p_2}[log, wit]$-membership.

As for the $\mathbf{FP}^{\Sigma^p_2}[log, wit]$-hardness part, in the proof of $\mathbf{FP}^{\Sigma^p_2}[log, wit]$-hardness for Theorem 14-(ii) each put-back set PB for the spurious answer set $\hat{I} = \emptyset$ for $A = \emptyset$ satisfies $omit(\Pi, \mathscr{A} \setminus PB) = \emptyset$, and is thus by Proposition 9 refinement-safe faithful. As the smallest size PB sets correspond to the maximum size $A' = \mathscr{A} \setminus PB$ sets, the $\mathbf{FP}^{\Sigma^p_2}[log, wit]$-hardness follows, even for $A_0 = \emptyset$. □

We remark that without refinement safety, the problem is likely to be more complex: deciding whether an abstraction is faithful is Π^p_2-complete, and this question is trivially reducible to this problem.

5 Refinement Using Debugging

Over-approximation of a program unavoidably introduces spurious answer sets, which makes it necessary to have an abstraction refinement method. We show how to employ an ASP debugging approach in order 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.* $\hat{I}$.

We use a meta-level debugging language (Brain et al., 2007) which is based on a tagging technique that allows one to control the building of answer sets and to manipulate the evaluation of the program. This is a useful technique for our need to shift the focus from “debugging the original program" to “debugging the inconsistency caused by the spurious answer set". We alter the meta-program, in a way that hints for refining the abstraction can be obtained. Through debugging, some of the atoms are determined as *badly omitted*, and by adding them back in the refinement the spurious answer set can be eliminated.

5.1 Debugging Meta-Program

The meta-program constructed by `spock` (Brain et al., 2007) introduces *tags* to control the building of answer sets. Given a program Π over $\mathscr{A}$ and a set $\mathscr{N}$ of names for all rules in Π, it creates an enriched alphabet $\mathscr{A}^+$ obtained from $\mathscr{A}$ by adding atoms such as $ap(n_r), bl(n_r), ok(n_r), ko(n_r)$ where $n_r \in \mathscr{N}$ 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. We omit the atoms $ok(n_r)$, as they are not needed. The (altered) meta-program that is created is as follows.

Definition 8 *Given Π, the program $\mathscr{T}_{meta}[\Pi]$ consists of the following rules for $r \in \Pi, \alpha_1 \in B^+(r), \alpha_2 \in B^-(r)$:*

$$H(r) \leftarrow ap(n_r), not\ ko(n_r).$$
$$ap(n_r) \leftarrow B(r).$$
$$bl(n_r) \leftarrow not\ \alpha_1.$$
$$bl(n_r) \leftarrow not\ not\ \alpha_2.$$

Here the last rules uses double (nested) negation *not not* α_2 (Lifschitz et al., 1999), which in the reduct w.r.t. an interpretation I is replaced by $\top$ if $I \models \alpha_2$ and by $\perp$ otherwise. The role of $ko(r)$ is to avoid the application of the rule $H(r) \leftarrow ap(r), not\ ko(r)$ if necessary. We use it for the rules that are changed due to some omitted atom in the body.

Abnormality atoms are introduced to indicate the cause of inconsistency: $ab_p(r)$ signals that rule r is falsified under some interpretation, $ab_c(\alpha)$ points out that α is true but has no support, and $ab_l(\alpha)$ indicates that α may be involved in a faulty loop (unfounded or odd).

Definition 9 *Given Π over $\mathscr{A}$, the following additional meta-programs are constructed:*

1. *$\mathscr{T}_P[\Pi]$: for all $r \in \Pi$ with $B(r) \cap A \neq \emptyset$ and $H(r) \neq \perp$:*

$$ko(n_r).$$
$$\{H(r)\} \leftarrow ap(n_r).$$
$$ab_p(n_r) \leftarrow ap(n_r), not\ H(r).$$

2. *$\mathscr{T}_C[\Pi, \mathscr{A}]$: for all $\alpha \in \mathscr{A} \setminus A$ with the defining rules $def(\alpha, \Pi) = \{r_1, ..., r_k\}$:*

$$\{\alpha\} \leftarrow bl(n_{r_1}), ..., bl(n_{r_k}).$$
$$ab_c(\alpha) \leftarrow \alpha, bl(n_{r_1}), ..., bl(n_{r_k}).$$

3. *$\mathscr{T}_A[\mathscr{A}]$: for all $\alpha \in \mathscr{A}$:*

$$\{ab_l(\alpha)\} \leftarrow not\ ab_c(\alpha).$$
$$\alpha \leftarrow ab_l(\alpha).$$

In $\mathscr{T}_C[\Pi, \mathscr{A}]$, we do not guess over the atoms A if all rules that have them in the head are blocked. This helps the search of a concrete interpretation for the partial/abstract interpretation by avoiding "bad" (i.e., non-supported) guesses of the omitted atoms. Notice that for the rules r_i with $H(r_i) = \alpha$ and empty body, we also put $bl(n_{r_i})$ so that $ab_c(\alpha)$ does not get determined, since one can always guess over α in Π.

Having $ab_l(\alpha)$ indicates that α is determined through a loop, but it does not necessarily show that the loop is unfounded (as described through *loop formulas* in (Brain et al., 2007). By checking whether α only gets support by itself, the unfoundedness can be caught. In some cases, α could be involved in an odd loop that was disregarded in the abstraction due to omission, which requires an additional check.

5.2 Determining Bad-Omission Atoms

Whether or not Π is consistent, our focus is on debugging the cause of inconsistency introduced through checking for a spurious answer set $\hat{I}$, i.e., evaluating the program $\Pi \cup Q^{\overline{A}}_{\hat{I}}$ from Proposition 2. in Section 3.2. We reason about the inconsistency by inspecting the reason for having $\hat{I} \in AS(omit(\Pi, A))$ due to some modified rules.

Definition 10 *Let $r : \alpha \leftarrow B$ be a rule in Π such that $B \cap A \neq \emptyset$ and $\alpha \notin A$. The abstract rule $\hat{r} : \{\alpha\} \leftarrow m_A(B)$ in $omit(\Pi, A)$ introduces w.r.t. an abstract interpretation $\hat{I} \in AS(omit(\Pi, A))$*

(i) a spurious choice*, if $\hat{I} \models m_A(B)$ and $\hat{I} \models \overline{\alpha}$, i.e., $\hat{I} \not\models \alpha$, but some model I of $\Pi \setminus \{r\}$ exists s.t. $I|_{\overline{A}} = \hat{I}$ and $I \models B$.*

(ii) a spurious support*, if $\hat{I} \models m_A(B)$ and $\hat{I} \models \alpha$, but some model I of Π exists s.t. $I|_{\overline{A}} = \hat{I}$ and for all $r' \in def(\alpha, \Pi), I \not\models B(r')$.*

Any occurrence of the above cases shows that $\hat{I}$ is spurious. In case (i), due to $\hat{I} \not\models \alpha$, the rule r is not satisfied by I while I is a model of the remaining rules. In case (ii), an I that matches $\hat{I} \models \alpha$ does not give a supporting rule for α.

Definition 11 *Let $r : \alpha \leftarrow B$ be a rule in Π such that $B \cap A \neq \emptyset$. The abstract rule $\hat{r} = m_A(r)$ introduces a* spurious loop-behavior *w.r.t. $\hat{I}$, if some model I of Π exists s.t. $I|_{\overline{A}} = \hat{I}$ and $I \models r$, but α is involved in a loop that is unfounded or is odd, due to some $\alpha' \in A \cap B$.*

The need for reasoning about the two possible faulty loop behaviors is shown by the following examples.

Example 9 *Consider the programs Π_1, Π_2 and their abstractions $\widehat{\Pi}_1 = \widehat{\Pi}_{1\overline{\{a\}}}$, $\widehat{\Pi}_2 = \widehat{\Pi}_{2\overline{\{a,b\}}}$.*

Π_1	$\widehat{\Pi}_1$	Π_2	$\widehat{\Pi}_2$
$r1 : a \leftarrow b.$		$r1 : a \leftarrow b.$	
$r2 : b \leftarrow not\ c, a.$	$\{b\} \leftarrow not\ c.$	$r2 : b \leftarrow not\ a, c.$	
		$r3 : c.$	$c.$

The program Π_1 has the single answer set $\emptyset$, and omitting a creates a spurious answer set $\{b\}$ disregarding that b in unfounded. The program Π_2 is unsatisfiable due to the odd loop of a and b. When both atoms are omitted, this loop is disregarded, which causes a spurious answer set $\{c\}$.

Bad omission of atoms are then defined as follows.

Definition 12 (bad omission atoms) *An atom $\alpha \in A$ is a* bad-omission *w.r.t. a spurious answer set $\hat{I}$ of $omit(\Pi, A)$, if some rule $r \in \Pi$ with $\alpha \in B(r)$ exists s.t. $\hat{r} = m_A(r)$ introduces either (i) a spurious choice, or (ii) a spurious support or (iii) a spurious loop-behavior w.r.t. $\hat{I}$.*

Intuitively, for case (i) of Definition 10, as $\overline{\alpha}$ was decided due to choice in $H(\hat{r})$, we infer that the omitted atom which caused r to become a choice rule is a bad-omission. Also for case (ii), as α is decided with $\hat{I} \models B(\hat{r})$, we infer that the omitted atom that caused $B(r)$ to be modified is a bad-omission. As for case (iii), it shows that the modification made on r (either omission or change to choice rule) ignores an unfoundedness or an odd loop. Case (i) also catches issues that arise due to omitting a constraint in the abstraction.

We now describe how we determine when an omitted atom is a bad omission.

Definition 13 (bad omission determining program) *The bad omission determining program $\mathscr{T}_{badomit}$ is constructed using the abnormality atoms obtained from $\mathscr{T}_P[\Pi]$, $\mathscr{T}_C[\Pi,\mathscr{A}]$ and $\mathscr{T}_A[\mathscr{A}]$ as follows:*

1. *A bad omission is inferred if the original rule is not satisfied, but applicable (and satisfied) in the abstract program:*

$$badomit(X,type1) \leftarrow ab_p(R), absAp(R), modified(R), omittedAtomFrom(X,R).$$

2. *A bad omission is inferred if the original rule is blocked and the head is unsupported, while it is applicable (and satisfied) in the abstract program:*

$$\begin{aligned} badomit(X,type2) \leftarrow\, & bl(R), head(R,H), ab_c(H), absAp(R), changed(R), \\ & omittedAtomFrom(X,R). \end{aligned}$$

3. *A bad omission is inferred in case there is unfoundedness or an involvement of an odd loop, via an omitted atom:*

$$\begin{aligned} faulty(X) \leftarrow\, & ab_l(X), inOddLoop(X,X1), omittedAtom(X1). \\ faulty(X) \leftarrow\, & ab_l(X), inPosLoop(X,X1), omittedAtom(X1). \\ badomit(X1,type3) \leftarrow\, & faulty(X), head(R,X), modified(R), absAp(R), \\ & omittedAtomFrom(X1,R). \end{aligned}$$

where $absAp(r)$ is an auxiliary atom to keep track of which original rule becomes applicable with the remaining non-omitted atoms for the abstract interpretation, $changed(r)$ shows that r is changed to a choice rule in the abstraction, and $modified(r)$ shows that r is either changed or omitted in the abstraction.

For defining *type3*, we check for loops using the encoding in (Syrjänen, 2006) and determine *inOddLoop* and (newly defined) *inPosLoop* atoms of Π.

The cases for *type2* and *type3* introduce as bad omissions the omitted atoms of all the rules that add to $ab_c(H)$ being true, or of all rules that have X in the head for $ab_l(X)$, respectively. Modifying *badomit* determination to have a choice over such rules to be refined (and their omitted atoms to be *badomit*) and minimizing the number of *badomit* atoms reduces the number of added back atoms in a refinement step, at the cost of increasing the search space.

In order to avoid the guesses of ab_l for omitted atoms even if there is no faulty loop behavior related with them (i.e., this is not the cause of inconsistency of $\hat{I}$), we add the constraint $\leftarrow ab_l(X), not\ someFaulty$.

With all this in place, the program for debugging a spurious answer set is composed as follows.

Definition 14 (spurious answer set debugging program) *For an abstract answer set $\hat{I}$, we shall denote by $\mathscr{T}[\Pi,\hat{I}]$ the program $\mathscr{T}_{meta} \cup \mathscr{T}_P[\Pi] \cup \mathscr{T}_C[\Pi,\mathscr{A}] \cup \mathscr{T}_A[\mathscr{A}] \cup \mathscr{T}_{badomit} \cup Q_{\hat{I}}^{\overline{A}}$.*

From the answer sets of $\mathscr{T}[\Pi,\hat{I}]$, we can see bad omissions and their types.

Example 10 *For the following program Π, $\hat{I}=\{b\}$ is a spurious answer set of the abstraction for $A=\{a,d\}$:*

Π	$\widehat{\Pi}_{\overline{a,d}}$
$r1: c \leftarrow not\ d.$	$\{c\}.$
$r2: d \leftarrow not\ c.$	
$r3: a \leftarrow not\ d, c.$	
$r4: b \leftarrow a.$	$\{b\}.$

$\mathscr{T}[\Pi,\hat{I}]$ *gives the answer set* $\{ap(r2),\ bl(r1),\ bl(r4),\ bl(r3),\ ab_c(b),\ badomit(a, type2)\}$.

The next example shows the need for reasoning about the disregarded positive loops and odd loops, due to omission.

Example 11 (Example 9 continued) *Recall that the program Π_1 has an unfounded loop between a and b, and the abstraction $\widehat{\Pi}_1=\widehat{\Pi}_{1\overline{\{a\}}}$ has the spurious answer set $\{b\}$. The program $\mathscr{T}[\Pi_1,\{b\}]$ now yields $inPosLoop(b,a)$, $ap(r1), ap(r2), ab_l(b), badomit(a, type3)$. Omitting from the program Π_2 the loop atoms a, b causes the spurious answer set $\{c\}$. Accordingly, $\mathscr{T}[\Pi_2,\{c\}]$ yields $ap(r3), inOddLoop(b,a)$, $ab_l(b)$, $ap(r1)$, $bl(r2), badomit(a, type3)$, as desired.*

The following result shows that $\mathscr{T}[\Pi,\hat{I}]$ flags in its answer sets always bad omission of atoms, which can be utilized for refinement.

Proposition 16 *If $\hat{I}$ is spurious, then for every answer set $S \in AS(\mathscr{T}[\Pi,\hat{I}])$, $badomit(\alpha, type\, i) \in S$ for some $\alpha \in A$ and $i \in \{1,2,3\}$.*

Proof. Let $\hat{I}$ be a spurious abstract interpretation. Then by Proposition 2 the program $\Pi \cup Q^{\overline{A}}_{\hat{I}}$ is unsatisfiable. We focus on debugging the cause of inconsistency introduced by $Q^{\overline{A}}_{\hat{I}}$. This inconsistency can either be due to (i) an unsatisfied rule, (ii) an unsupported atom, or (iii) an unfounded support from a loop.

Case (i): let r be an unsatisfied rule in $\Pi \cup Q^{\overline{A}}_{\hat{I}}$. This means that the constraints in $Q^{\overline{A}}_{\hat{I}}$ is causing $H(r)$ to be false while $B(r)$ is satisfied. So in $\mathscr{T}[\Pi,\hat{I}]$, according to its definition, $ab_p(r)$ becomes true. The remainder of $B(r)$ after the omission also holds true, i.e., $absAp(r) = true$. Thus, the definition of $badomit(\_, type1)$ is able to catch this case.

Case (ii): let a be an unsupported atom in $\Pi \cup Q^{\overline{A}}_{\hat{I}}$. First, a must be from $\overline{A}$ because $Q^{\overline{A}}_{\hat{I}}$ only restricts the determined value of these remainder atoms. In $\mathscr{T}[\Pi,\hat{I}]$, according to its definition, $ab_c(a)$ becomes true. The value of a is determined from a changed rule r with $H(r)=a$, due to the remainder of $B(r)$ becoming satisfiable with $\hat{I}$, while originally it is not, i.e. $bl(R) = false$ and $absAp(R) = true$. The definition of $badomit(\_, type2)$ is able to catch this case and choose the omitted atom in the body to be bad.

Case (iii): if the abstracted version of rule r becomes applicable in $omit(\Pi,A)$, while originally the truth-value of $H(r)$ requires to get support via a loop for a model I that matches $\hat{I}$, then $ab_l(a)$ catches this case. If the loop is the only support that $H(r)$ can obtain, and if this loop is unfounded or odd, then this should be reflected in the abstraction of r. Thus, by $badomit(\_, type3)$ this case is caught. $\square$

The badly omitted atoms $A_o \subseteq A$ w.r.t a spurious $\hat{I} \in AS(omit(\Pi,A))$ are added back to refine m_A. If $\hat{I}$ still occurs in the refined program $omit(\Pi, A \setminus A_o)$, i.e., some $\hat{I}' \in AS(omit(\Pi, A \setminus A_o))$ with $\hat{I}'|_{\overline{A}} = \hat{I}$ exists, then $\mathscr{T}[\Pi,\hat{I}']$ finds another possible bad omission. In the worst case, all omitted atoms A are put back to eliminate $\hat{I}$.

Corollary 17 *After at most $|A|$ iterations of the program, the spurious answer set will no longer occur.*

Adding back a badly omitted atom may cause a previously omitted rule to appear as a changed rule in the refined program. Due to this choice rule, the spurious answer set might not get eliminated. To give a (better) upper bound for the number of required iterations in order to eliminate a spurious answer set, a trace of the dependencies among the omitted rules is needed.

The *rule dependency graph* of Π, denoted $G_{\Pi}^{rule} = (V,E)$, shows the positive/negative dependencies similarly as in G_{Π}, but at a rule-level, where the vertices V are rules $r \in \Pi$ and an edge from r to r' exists in E if the $H(r') \in B^{\pm}(r)$ holds, which is negative if $H(r') \in B^{-}(r)$ and positive otherwise. For a set A of atoms, n_A denotes the maximum length of a (non-cyclic) path in G_{Π}^{rule} from some rule r with $B(r) \cap A \neq \emptyset$ backwards through rules r' with $H(r') \cap A \neq \emptyset$. The number n_A shows the maximum level of direct or indirect dependency between omitted atoms and their respective rules.

Proposition 18 *Given a program Π, a set A of atoms, and a spurious $\hat{I} \in AS(omit(\Pi,A))$, after at most n_A iterations of finding a bad omission with $\mathscr{T}[\Pi,\hat{I}]$ and refinement, no abstract answer set matching $\hat{I}$ will occur.*

Proof. Let r_0 be a rule with $\alpha \in B(r_0) \cap A$ that is changed to a choice rule due to m_A. Let $r_0, r_1, \ldots, r_{n_A}$ be a dependency path in G_{Π}^{rule} where $H(r_i) \cap A \neq \emptyset$ and $B(r_i) \cap A \neq \emptyset$, $0 \leq i < n_A$. Let $\hat{I} \in AS(omit(\Pi, A))$, assume r_0 has spurious behavior w.r.t. $\hat{I}$, and, w.l.o.g. assume $\hat{I} \models B(r_i) \setminus A$ for all $i \leq n_A$.

Due to inconsistency via r_0, $badomit(\alpha) \in AS(\mathscr{T}[\Pi,\hat{I}])$. For $A' = A \setminus \{\alpha\}$, $m_{A'}(r_0)$ is unchanged, while $m_{A'}(r_1)$ becomes a choice rule (with $n_A - 1$ dependencies left). Thus, some $I' \in AS(omit(\Pi,A'))$ with $I|_{\overline{A}}' = \hat{I}$ can still exist. Since r_1 introduces spuriousness w.r.t. I', there is $badomit(\alpha') \in AS(\mathscr{T}[\Pi,I'])$ for $\alpha' \in B(r_1) \cap A'$.

By iterating this process n_A times, all omitted rules on which r_0 depends are traced and eventually no abstract answer set matching $\hat{I}$ occurs. □

We remark that in case more than one dependency path $r_0, \ldots, r_{n_A}$ with several rules causing inconsistencies exists, the returned set of *badomit*s from $\mathscr{T}[\Pi,\hat{I}]$ allows one to refine the rules in parallel).

Recall that Proposition 8 ensures that adding back further omitted atoms will not reintroduce a spurious answer set. Further heuristics on the determination of bad omission atoms can be applied in order to ensure that a spurious answer set is eliminated in one step.

6 Application: Catching Unsatisfiability Reasons of Programs

In this section, we consider as an application case the use of abstraction in finding a cause of unsatisfiability for an ASP program. To this end, we first introduce the notion of blocker sets for understanding which of the atoms are causing the unsatisfiability.

After describing the implementation, we report about our experiments where the aim was to observe the use of abstraction and refinement for achieving an over-approximation of a program that is still unsatisfiable and to compute the $\subseteq$-minimal blockers of the programs, which projects away the part that is unnecessary for the unsatisfiability.

6.1 Blocker Sets of Unsatisfiable Programs

If a program Π has no answer sets, we can obtain by omitting sufficiently many atoms from it an abstract program that has some abstract answer set. By Proposition 3-(iv), any such answer set will be spurious. On the other hand, as long as the abstracted program has no answer sets, by Proposition 3-(iii) also the original program Π has no answer set. This motivates us to use omission abstraction in order to catch a "real" cause of inconsistency in a program. To this end, we introduce the following notion.

Definition 15 *A set $C \subseteq \mathscr{A}$ of atoms is an* (answer set) blocker set *of Π, if $AS(omit(\Pi, \mathscr{A} \setminus C)) = \emptyset$.*

In other words, when we keep the set C of atoms and omit the rest from Π to obtain the abstract program Π', then the latter is still unsatisfiable. This means the atoms in C are *blocking* the occurrence of answer sets: no answer set is possible as long as all these atoms are present in the program, regardless of how the omitted atoms will be evaluated in building an answer set.

Example 12 (Example 3 continued) *Modify Π by changing the last rule to $b \leftarrow not\ b.$, in order to have a program Π' which is unsatisfiable. Omitting the set $A = \{d\}$ from Π' creates the abstract program $\widehat{\Pi}'_{\overline{\{d\}}}$ which is still unsatisfiable. Thus, the set $C = \mathscr{A} \setminus A = \{a,b,c\}$ is a blocker set of Π'. This is similar for omitting the set $A = \{a,c\}$, which then causes to have $C = \{d,b\}$ as a blocker set of Π'.*

Π'	$\widehat{\Pi}'_{\overline{\{d\}}}$	$\widehat{\Pi}'_{\overline{\{a,c\}}}$
$c \leftarrow not\ d.$	$\{c\}.$	
$d \leftarrow not\ c.$		$\{d\}$
$a \leftarrow not\ b, c.$	$a \leftarrow not\ b, c.$	
$b \leftarrow not\ b.$	$b \leftarrow not\ b.$	$b \leftarrow not\ b.$
unsatisfiable	*unsatisfiable*	*unsatisfiable*

Notice that $C = \mathscr{A}$, i.e., no atom is omitted, is trivially a blocker set if Π is unsatisfiable, while $C = \emptyset$, i.e., all atoms are omitted, is never a blocker set since $AS(omit(\Pi, \mathscr{A})) = \{\emptyset\}$.

We can view a blocker set as an *explanation* of unsatisfiability; by applying Occam's razor, simpler explanations are preferred, which in pure logical terms motivates the following notion.

Definition 16 *A blocker set $C \subseteq \mathscr{A}$ is* $\subset$-minimal*, if for all $C' \subset C$, $AS(omit(\Pi, \mathscr{A} \setminus C')) \neq \emptyset$.*

By Proposition 9, in order to test whether a blocker set C is minimal, we only need to check whether for no $C' = C \setminus \{c\}$, for $c \in C$, the abstraction $omit(\Pi, \mathscr{A} \setminus C')$ has an answer set. That is, for a minimal blocker set C, we have that $\mathscr{A} \setminus C$ is a *maximal unsatisfiable abstraction*, i.e., a maximal set of atoms that can be omitted while keeping the unsatisfiability of Π.

Example 13 (Example 12 continued) *The program Π' has the single minimal blocker set $C = \{b\}$. Indeed, the rule $r5 : b \leftarrow not b$ does not admit an answer set. Thus, every blocker set must contain b, and C is the smallest such set.*

We remark that the atoms occurring in the blocker sets are intuitively the ones responsible for the unsatisfiability of the program. In order to observe the reason of unsatisfiability, one has to look at the remaining abstract program. For this, we consider the notion of *blocker rule set* associated with a blocker set C, which are the rules that remain in $omit(\Pi, \mathscr{A} \setminus C)$. For example, the programs $\Pi', \widehat{\Pi}'_{\overline{\{d\}}}$ and $\widehat{\Pi}'_{\overline{\{a,c\}}}$ in Example 12 contain the blocker rule sets associated with $\{a,b,c,d\}, \{a,b,c\}$ and $\{b,d\}$, respectively. Here, the abstract

Figure 3: Program for 2-colorability (adapted from the coloring encoding in the ASP Competition 2013)

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

programs contain choice rules due to the omission in the body, and the unsatisfiability of the programs shows that the evaluation of the respective rule does not make a difference for unsatisfiability. In other words, whether these rules are projected to the original rules by removing the choice, e.g. $\{c\}$. in $\widehat{\Pi}'_{\overline{\{d\}}}$ gets changed to c., or whether they are converted into constraints, e.g. $\leftarrow not\ c$, the program will still be unsatisfiable.

Example 12 illustrated a simple reason for unsatisfiability. However, the introduced notion is also able to capture more complex reasons of unsatisfiability that involve multiple rules related with each other, which is illustrated in the next example.

Example 14 (Graph coloring) *Consider coloring the graph shown in Figure 1 with two colors green and red. Due to the clique formed by the nodes* $1,2,3$*, it is not 2-colorable. A respective encoding is shown in Figure 6.1, which for the given graph reduces by grounding and elimination of facts to the following rules, where* $n \in \{1,\dots,9\}$*, and* $c, c_1, c_2 \in \{red, green\}$*:*

$$
\begin{aligned}
&\{chosenColor(n,c)\}.\\
&colored(n) \leftarrow chosenColor(n,c).\\
&\leftarrow not\ colored(n).\\
&\leftarrow chosenColor(n,c_1), chosenColor(n,c_2), c_1 \neq c_2.\\
&\leftarrow chosenColor(n_1,c), chosenColor(n_2,c). \qquad nodes\ n_1,n_2\ are\ adjacent
\end{aligned}
$$

Omitting a node n in the graph means to omit all ground atoms related to n; omitting all nodes except $1,2,3$ *gives us a blocker set with the corresponding blocker rule set shown in Figure 4. This abstract program is unsatisfiable and omitting further atoms in the abstraction yields spurious satisfiability. The set of atoms that remain in the program is actually the minimal blocker set for this program. We can also observe the property of unsatisfiable programs being refinement-safe faithful (Proposition 9), as refining the shown abstraction by adding back atoms relevant with the other nodes will still yield unsatisfiable programs.*

For the introduced notions of blocker sets, the below result follows from Theorem 15.

Corollary 19 *Computing (i) some* $\subseteq$*-minimal respectively (ii) some smallest size blocker* $C \subseteq \mathscr{A}$ *for a given program* Π *is (i) in* $\mathbf{FP}^{\mathbf{NP}}$ *and* $\mathbf{FP}^{\mathbf{NP}}_{\parallel}$*-hard respectively (ii)* $\mathbf{FP}^{\Sigma_2^P}[log, wit]$*-complete.*

The membership follows for the case that Π has no answer sets, and the hardness by the reduction in the proof of Theorem 15.

Figure 4: Blocker rule set for 2-colorability of Figure 1(a)

$\{chosenColor(1,red)\}.$	$\leftarrow not\ colored(1).$
$\{chosenColor(2,red)\}.$	$\leftarrow not\ colored(2).$
$\{chosenColor(3,red)\}.$	$\leftarrow not\ colored(3).$
$\{chosenColor(1,green)\}.$	$\leftarrow chosenColor(1,red), chosenColor(1,green).$
$\{chosenColor(2,green)\}.$	$\leftarrow chosenColor(2,red), chosenColor(2,green).$
$\{chosenColor(3,green)\}.$	$\leftarrow chosenColor(3,red), chosenColor(3,green).$
$colored(1) \leftarrow chosenColor(1,red).$	$\leftarrow chosenColor(2,red), chosenColor(1,red).$
$colored(2) \leftarrow chosenColor(2,red).$	$\leftarrow chosenColor(3,red), chosenColor(1,red).$
$colored(3) \leftarrow chosenColor(3,red).$	$\leftarrow chosenColor(3,red), chosenColor(2,red).$
$colored(1) \leftarrow chosenColor(1,green).$	$\leftarrow chosenColor(2,green), chosenColor(1,green).$
$colored(2) \leftarrow chosenColor(2,green).$	$\leftarrow chosenColor(3,green), chosenColor(1,green).$
$colored(3) \leftarrow chosenColor(3,green).$	$\leftarrow chosenColor(3,green), chosenColor(2,green).$

6.2 Implementation

The experiments have been conducted with a tool[2] that we have implemented according to the described method. It uses Python, Clingo (Gebser et al., 2011) and the meta-program output of the Spock debugger (Brain et al., 2007).

The procedure for the abstraction and refinement method is shown in Algorithm 1. Given a program Π and a set A_{init} of atoms to be omitted, first the abstract program $\Pi' = omit(\Pi, A_{init})$ is constructed (Line 2). If the abstract program is unsatisfiable, the program and the set of omitted atoms are returned (Line 13). Otherwise, an answer set $I \in AS(omit(\Pi, A_{init}))$ is computed. In the implementation, the first answer set is picked. In order to check whether I is concrete, the meta-program $\Pi_{debug} = \mathscr{T}[\Pi, \hat{I}]$ as described in Section 5 is constructed (Line 5). Then a search over the answer sets of $\mathscr{T}[\Pi, \hat{I}]$ for a minimum number of *badomit* atoms is carried out (Line 6). If an answer set with no *badomit* atoms exists, then this shows that I is concrete, and the abstract program and the set of omitted atoms are returned (Line 8). Otherwise, the set of omitted atoms is refined by removing the atoms that are determined as badly omitted, and a new abstract program is constructed with the refined abstraction A'. This loop continues until either the abstract program Π' constructed at Line 11 is unsatisfiable or its first answer set is concrete.

Figure 6 shows the implemented system according to Algorithm 1 with the respective components. The arcs model both control and data flow within the tool. The workflow of the tool is as follows. First, the input program Π and the set A of atoms to be omitted is read. Then the *controller* calls the *abstraction creator* component which uses Π and A to create the abstract program $\widehat{\Pi}_A$ $\boxed{1}$. The controller then calls the *ASP Solver* to get an answer set of $\widehat{\Pi}_A$ $\boxed{2}$. If the solver finds no answer set, the controller outputs the abstract program and the set of omitted atoms. Otherwise, it calls the *refinement* component with the abstract answer set $\hat{I}$ to decide whether or not to refine the abstraction $\boxed{3}$. The refinement component calls the *checker creator* $\boxed{4}$ to create $\mathscr{T}[\Pi, \hat{I}]$, which uses Spock $\boxed{5}$, and then calls the ASP solver to check whether $\hat{I}$ is concrete $\boxed{6}$. If not, i.e., when $\hat{I}$ is spurious, it refines the abstraction by updating A $\boxed{7}$. Otherwise, the controller returns the outputs.

The computation of a $\subseteq$-minimal blocker set of an unsatisfiable program, given an initial set of omission atoms A, is shown in Algorithm 2; it derives from computing some $\subseteq$-minimal put-back set (Theorem 14),

[2] www.kr.tuwien.ac.at/research/systems/abstraction

```
Algorithm 1: Abs&Ref
    Input: $\Pi$, $A_{init}$
    Output: $\Pi' = omit(\Pi, A'), A'$
1   $A' = A_{init}$;
2   $\Pi' = constructAbsProg(\Pi, A')$;
3   while $AS(\Pi') \neq \emptyset$ do
4       Get $I \in AS(\Pi')$;
5       $\Pi_{debug} = constructDebugProg(\Pi, A', I)$;
6       $S = getASWithMinBadOmit(\Pi_{debug})$;
7       if $S|_{badomit} = \emptyset$ then /* I concrete */
8           return $\Pi', A'$
9       else /* refine the abstraction */
10          $A' = A' \setminus S|_{badomit}$;
11          $\Pi' = constructAbsProg(\Pi, A')$;
12  /* reached an unsatisfiable $\Pi'$ */
13  return $\Pi', A'$
```

```
Algorithm 2: ComputeMinBlocker
    Input: $\Pi$, $\mathscr{A}$, $A$ s.t. $AS(\Pi, A) = \emptyset$
    Output: a $\subseteq$-minimal blocker set $C_{\min} \subseteq \mathscr{A} \setminus A$
1   forall the $\alpha \in \mathscr{A} \setminus A$ do
2       $\Pi' = constructAbsProg(\Pi, \{\alpha\})$;
3       if $AS(\Pi') = \emptyset$ then
4           $A = A \cup \{\alpha\}$;
5           $\Pi = \Pi'$;
6   return $C_{\min} = \mathscr{A} \setminus A$
```

Figure 5: Algorithms for Abstraction and Refinement (left) an Minimal Blocker Computation (right)

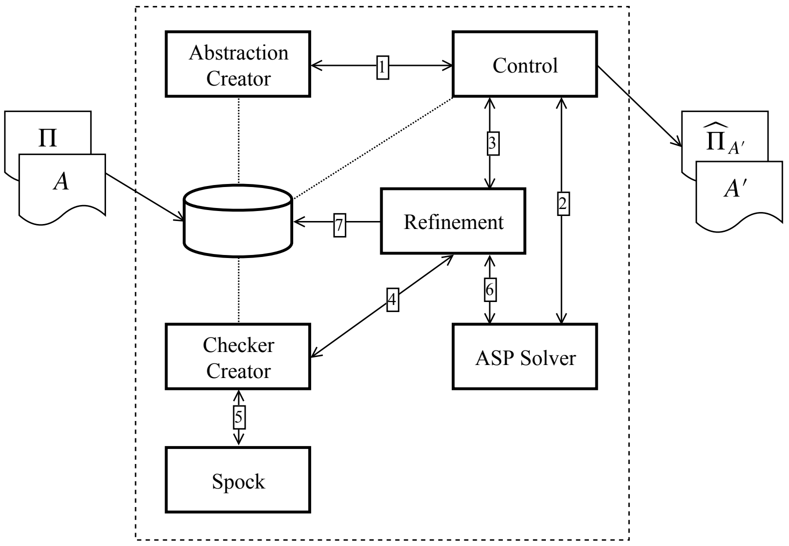

Figure 6: System architecture of the implementation

by taking into account that minimal blocker sets amount to minimal put-back set for unsatisfiability. The procedure checks whether omitting an atom $\alpha \in \mathscr{A} \setminus A$ from Π preserves unsatisfiability. If yes, the atom is added to A and the search continues from the newly constructed abstract program $omit(\Pi, \{\alpha\})$. Once all the atoms are examined, the atoms that are chosen not to be omitted $\mathscr{A} \setminus A$ is a $\subseteq$-minimal blocker set, provided that $AS(\Pi, A)$ is unsatisfiable.

6.3 Experiments

In our experiments, we wanted to observe the use of abstraction in catching the part of the program which causes unsatisfiability. We aimed at studying how the abstraction and refinement method behaves in different benchmarks in terms of the computed final abstractions and the needed refinement steps, when starting with an initial omission of a random set of atoms. For the refinement step, we expected the search for the answer set with minimum number of *badomit* atoms to be difficult, and thus wanted to investigate whether different minimizations over the *badomit* atom number makes a difference in the reached final abstractions.

Additionally, we were interested in computing the $\subseteq$-minimal blocker sets of the programs and observing the difference in size of the $\subseteq$-minimal blocker sets depending on the problems. For finding $\subseteq$-minimal blocker sets, we additionally compared a *top-down* method to a *bottom-up* method, to see their effects on the quality of the resulting $\subseteq$-minimal blocker sets. The top-down method proceeds by calling the function *ComputeMinBlocker* with the original program Π, $\mathscr{A}$ and $A = \{\}$, so that the search for a $\subseteq$-minimal blocker set starts from the top. The bottom-up method initially chooses a certain percentage of the atoms to omit, A_{init}, and calls the function *Abs&Ref* with Π and A_{init} to refine the abstraction and find an unsatisfiable abstract program, $omit(\Pi, A_{final})$. Then, a search for $\subseteq$-minimal blocker sets is done, with the remaining atoms, by calling the function *ComputeMinBlocker* with $omit(\Pi, A_{final})$, $\mathscr{A}$ and A_{final}. We wanted to observe whether there are cases where the bottom-up method helps in reaching better quality $\subseteq$-minimal blocker sets that have smaller size than those obtained with the top-down method.

6.3.1 Benchmarks

We considered five benchmark problems with a focus on the unsatisfiable instances. Two of the problems are based on graphs, two are scheduling and planning problems, respectively, and the fifth one is a subset selection problem.

Graph Coloring (GC). We obtained the generator for the graph coloring problem[3] that was submitted to the ASP Competition 2013 (Alviano et al., 2013), and we generated 35 graph instances with node size varying from 20 to 50 with edge probability 0.2 to 0.6, which are not 2 or 3-colorable. The respective colorability tests are added as superscripts to GC, i.e, GC^2, GC^3.

Abstract Argumentation (AA). Abstract argumentation frameworks are based on graphs to represent and reason about arguments. The abstract argumentation research community has a broad collection of benchmarks with different types of graph classes, which are also being used in competitions (Gaggl et al., 2016). We obtained the Watts-Strogatz (WS) instances (Watts and Strogatz, 1998) that were generated by (Cerutti et al., 2016) and are unsatisfiable for existence of so called stable extensions.[4] We focused on the unsatisfiable (in total 45) instances with 100 arguments (i.e., nodes) where each argument is connected (i.e., has an edge) to its $n \in \{6, 12, 18\}$ nearest neighbors and it is connected to the remaining arguments with a probability $\beta \in \{0.10, 0.30, 0.50, 0.70, 0.90\}$.

Disjunctive Scheduling (DS). As a non-graph problem, we considered the task scheduling problem from the ASP Competition 2011[5] and generated 40 unsatisfiable instances with $t \in \{10, 20\}$ tasks within $s \in \{20, 30\}$ time steps, where $d \in \{10, 20\}$ tasks are randomly chosen to not to have overlapping schedules.

[3] `www.mat.unical.it/aspcomp2013/GraphColouring`
[4] `www.dbai.tuwien.ac.at/research/project/argumentation/systempage/Data/stable.dl`
[5] `www.mat.unical.it/aspcomp2011`

Strategic Companies (SC). We considered the strategic companies problem with the encoding and simple instances provided in (Eiter et al., 1998). In order to achieve unsatisfiability, we added a constraint to the encoding that forbids having all of the companies that produce one particular product to be strategic. *SC* is a canonic example of a disjunctive program that has presumably higher computational cost than normal logic programs, and no polynomial time encoding into program such program is feasible. We have thus split rules with disjunctive heads, e.g., $a \vee b \leftarrow c$, into choice rules $\{a\} \leftarrow c; \{b\} \leftarrow c$ at the cost of introducing spurious guesses and answer sets. The resulting split program can be seen as an over-approximation of the original program, and thus causes for unsatisfiability of the split program as approximative causes for unsatisfiability of the original program.

15-puzzle (PZ). Inspired from the Unsolvability International Planning Competition[6], we obtained the ASP encoding for the Sliding Tiles problem from the ASP Competition 2009 [7], which is named as 15-puzzle. We altered the encoding in order to avoid having cardinality constraints in the rules, and to make it possible to also solve non-square instances. We used the 20 unsolvable instances from the planning competition, which consist of 10 instances of 3x3 and 10 instances of 4x3 tiles.

The collection of all encodings and benchmark instance can be found at `http://www.kr.tuwien.ac.at/research/systems/abstraction/`

6.3.2 Results

The tests were run on an Intel Core i5-3450 CPU @ 3.10GHz machine using Clingo 5.3, under a 600 secs time and 7 GB memory limit. The initial omission, A_{init}, is done by choosing randomly 50%,75% or 100% of the nodes in the graph problems GC, AA, of the tasks in DS, of the companies in SC, and of the tiles in PZ, as well as by omitting all the atoms related with the chosen objects. We show the overall average of 10 runs for each instance in Figure 7.

The first three rows under each category show the bottom-up approach for 50%,75%, and 100% initial omission, respectively. The columns $|A_{init}|/|\mathscr{A}|$ and $|A_{final}|/|\mathscr{A}|$ show the ratio of the initial omission set A_{init} and the final omission set A_{final} that achieves unsatisfiability after refining A_{init} (with shown number of refinement steps and time). The second part of the columns is on the computation of a $\subseteq$-minimal blocker set $C_{\min}$. For the bottom-up approach, the search starts from A_{final} while for the top-down approach, it starts from $\mathscr{A}$. In each refinement step, the number of determined *badomit* atoms are minimized to be at most $|A|/2$; Figure 8 shows results for different upper limits and its full minimization.

Figure 7 shows that, as expected, there is a minimal part of the program which contains the reason for unsatisfiability of the program by projecting away the atoms that are not needed (sometimes more than 90% of all atoms). Observe that when 100% of the objects in the problems are omitted, refining the abstraction until an unsatisfiable abstract program takes the most time. This shows that a naive way of starting with an initial abstraction by omitting every relevant detail is not efficient in reaching an unsatisfiable abstract program. We can observe that larger A_{final} results in having less time spent in computing $\subseteq$-minimal blocker sets, as a smaller number of atoms must be checked. Additionally, with a bottom-up method it is possible to reach a $\subseteq$-minimal blocker set which is smaller in size than the ones obtained with the top-down method.

The graph coloring benchmarks ($GC^{2,3}$) show that more atoms are kept in the abstraction to catch the non-3-colorability than the non-2-colorability, which matches our intuition. For example, in GC^2 omitting 50% of the nodes (49% of the atoms in A_{init}) already reaches an unsatisfiable program, since no atoms were

[6] `https://unsolve-ipc.eng.unimelb.edu.au/`

[7] `https://dtai.cs.kuleuven.be/events/ASP-competition`

Figure 7: Experimental results for the base case (i.e., with upper limit on *badomit* # per step). The three entries in a cell, e.g., 0.49 / 0.74 / 1.00 in cell (GC^2, $\frac{|A_{init}|}{|\mathscr{A}|}$), are for 50% / 75% / 100% initial omission.

Π	$\frac{\lvert A_{init}\rvert}{\lvert\mathscr{A}\rvert}$	$\frac{\lvert A_{final}\rvert}{\lvert\mathscr{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathscr{A}\rvert}$	t (sec)
GC^2	0.49	0.49	0.02	0.81	0.10	0.80
	0.74	0.63	0.51	1.13	0.10	0.51
	1.00	0.18	3.03	3.60	0.10	1.63
	top-down				0.10	2.30
GC^3	0.49	0.40	0.82	1.83	0.17	1.68
	0.72	0.31	2.46	5.87	0.16	2.04
	1.00	0.11	4.18	6.54	0.17	3.47
	top-down				0.16	4.32
AA	0.50	0.19	3.70	7.20	0.38	8.90
	0.75	0.20	4.19	8.41	0.37	8.67
	1.00	0.01	2.00	4.07	0.38	11.74
	top-down				0.38	11.75
DS	0.50	0.39	1.62	3.36	0.10	1.89
	0.72	0.40	3.49	6.77	0.09	2.09
	1	0.45	4.9	9.57	0.07	1.99
	top-down				0.09	4.15
SC	0.49	0.48	0.03	0.59	0.10	0.34
	0.74	0.42	0.65	1.14	0.10	0.41
	1.00	0.43	1.00	2.65	0.11	0.40
	top-down				0.12	0.82
PZ	0.36	0.32	3.76	65.1	0.29	150.1
	0.54	0.45	8.47	154.1	0.27	103.7
	0.76	0.54	22.85	448.6	0.26	80.0
	top-down				0.30	281.4

added back in A_{final}. However, for GC^3 an average of only 9% of the omitted atoms were added back until unsatisfiability is caught.

For the GC^{2,3}, SC and PZ benchmarks, we can observe that omitting 50% of the objects ends up easily in reaching some unsatisfiable abstract program, with refinements of the abstractions being relatively small. For example, for GC^2 the size of A_{final} is the same as for A_{init}, and for PZ an average of only 4% of the atoms is added back in A_{final}. However, this behavior is not observed when initially omitting 75% of the objects. We can also observe that some problems (AA and PZ) have larger $\subseteq$-minimal blocker sets than others. This shows that these problems have a more complex structure than others, in the sense that more atoms are syntactically related with each other through the rules and have to be considered for obtaining the unsatisfiability.

Figure 8: Experimental results with different upper limits on *badomit* #. The three entries in a cell, e.g., 0.21 / 0.24 / 0.23 in cell (AA, $\frac{|A_{final}|}{|\mathscr{A}|}$) of *badomit*# $\leq |\mathscr{A}|/5$, are for 50% / 75% / 100% initial omission.

	badomit# $\leq \lvert\mathscr{A}\rvert/5$					*badomit*# $\leq \lvert\mathscr{A}\rvert/10$				
Π	$\frac{\lvert A_{final}\rvert}{\lvert\mathscr{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathscr{A}\rvert}$	t (sec)	$\frac{\lvert A_{final}\rvert}{\lvert\mathscr{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathscr{A}\rvert}$	t (sec)
	0.21	4.84	9.49	0.37	8.93	0.23	6.90	13.59	0.36	8.69
AA	0.24	5.93	11.92	0.36	8.38	0.29	8.61	17.84	0.35	7.86
	0.23	5.87	11.93	0.36	8.88	0.33	10.27	22.30	0.34	7.36

	min_badomit#				
Π	$\frac{\lvert A_{final}\rvert}{\lvert\mathscr{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathscr{A}\rvert}$	t (sec)
	0.24	7.89	15.20	0.36	8.06
AA	0.30	10.65	34.10 (2)	0.34	7.06
	0.44	17.48	62.46 (1)	0.34	5.86

Badomit minimization In a refinement step, minimizing the number of *badomit* atoms gives the smallest set of atoms to put back. However, the minimization makes the search more difficult, hence may hit a timeout; e.g., no optimal solution for 45 nodes in *GC* was found in 10 mins. Figure 8 shows the results of giving different upper bounds on the number of *badomit* atoms and also applying the full minimization in the refinement for the *AA* instances. The numbers in the parentheses show the number of instances that reached a timeout. As more minimization is imposed, we can observe an increase in the size of the final omissions A_{final} and also a decrease in the size of the $\subseteq$-minimal blocker set. For example, for 75% initial omission, we can see that the size of the computed final omission increases from 0.20 (Figure 7) to 0.24, 0.29 and finally to 0.30. Also the size of the $\subseteq$-minimal blocker set decrease from 0.37 (Figure 7) to 0.36, 0.35 and finally to 0.34. As expected, adding the smallest set of *badomit* atoms back makes it possible to reach a larger omission A_{final} that keeps unsatisfiability (e.g., *min_badomit*# third row (100% A_{init}): A_{final} is 44% instead of 0.01% as in Figure 7). On the other hand, such minimization over the number of *badomit* atoms causes to have more refinement steps (Ref #) to reach some unsatisfiable abstract program, which also adds to the overall time.

The $\subseteq$-minimal blocker search algorithm relies on the order of the picked atoms. We considered the heuristics of ordering the atoms according to the number of rules in which each atom shows up in the body, and starting the minimality search by omitting the least occurring atoms. However, this did not provide better results than just picking an atom arbitrarily.

Sliding Tiles (15-puzzle) Studying the resulting abstract programs with $\subseteq$-minimal blockers showed that finding out whether the problem instance is unsolvable within the given time frame does not require to consider every detail of the problem. Omitting the details about some of the tiles still reaches a program which is unsolvable, and shows the reason for unsolvability through the remaining tiles. Figure 9 shows an instance from the benchmark, which is unsolvable in 10 steps. Applying omission abstraction achieves

Figure 9: Unsolvable sliding tiles problem instance

Initial State

0	3	2
8	5	4
1	6	7

Goal State

0	1	2
3	4	5
6	7	8

(a) Concrete problem

Initial state

0	3	*
*	5	4
*	*	*

Goal State

0	*	*
3	4	5
*	*	*

(b) Abstract problem

an abstract program that only contains atoms relevant with the tiles 0,3,4,5 and is still unsatisfiable; this matches the intuition behind the notion of pattern databases introduced in (Culberson and Schaeffer, 1998).

Summary The results show that the notion of abstraction is useful in computing the part of the problem which causes unsatisfiability, as all of the benchmarks contain a blocker set that is smaller than the original vocabulary. We observed that different program structures cause the $\subseteq$-minimal blocker sets to be different in size with respect to the respective original vocabulary size. Computation of these $\subseteq$-minimal blocker sets can sometimes result in smaller sizes with the bottom up approach. However, starting with an 100% initial omission to use the bottom-up approach appears to be unreasonable due to the time difference compared to the top-down approach, even though sometimes it computes $\subseteq$-minimal blocker atoms sets of smaller size. The abstraction & refinement approach can also be useful if there is a desire to find some (non-minimal) blocker, as most of the time, starting with an initial omission of 50% or 75% results in computing some unsatisfiable abstraction in few refinement steps.

We recall that our focus in this initial work is on the usefulness of the abstraction approach on ASP, and not on the scalability. However, we believe that further implementation improvements and optimization techniques should also make it possible to argue about efficiency.

7 Discussion

In this section, we first discuss possible extensions of the approach to more expressive programs, in particular to non-ground programs and to disjunctive logic programs, and we then address further aspects that may influence the solving behavior.

7.1 Non-Ground Case

In case of omitting atoms from non-ground programs, a simple extension of the method described above is to remove all non-ground atoms from the program that involve a predicate p that should be omitted. This, however, may require to introduce domain variables in order to avoid the derivation of spurious atoms. Specifically, if in a rule $r : \alpha \leftarrow B(r)$ a non-ground atom $p(V_1,\dots,V_n)$ that is omitted from the body shares some arguments, V_i, with the head α, then α is conditioned for V_i with a domain atom $dom(V_i)$ in the constructed rule, so that all values of V_i are considered.

Example 15 *Consider the following program* Π *with domain predicate int for an integer domain* $\{1,...,5\}$*:*

$$a(X_1,X_2) \leftarrow c(X_1), b(X_2). \tag{22}$$

$$d(X_1,X_2) \leftarrow a(X_1,X_2), X_1 \leq X_2. \tag{23}$$

In omitting $c(X)$, while rule (23) remains the same, rule (22) changes to

$$\{a(X_1,X_2) : int(X_1)\} \leftarrow b(X_2).$$

From Π and the facts $c(1),b(2)$, we get the answer set $\{c(1),b(2),\ a(1,2),d(1,2)\}$, and with $c(2),b(2)$ we get $\{c(2),b(2),\ a(2,2),d(2,2)\}$. After omitting $c(X)$, the abstract program with fact $b(2)$ has 32 answer sets. Among them are $\{b(2),a(1,2),d(1,2)\}$ and $\{b(2),\ a(2,2),d(2,2)\}$, which cover the original answer sets, i.e., each original answer set can be mapped to some abstract one.

For a more fine-grained omission, let the set A consist of the atoms $\alpha = p(c_1,\dots,c_k)$ and let $A_p \subseteq A$ denote the set of ground atoms with predicate p that we want to omit. Consider a k-ary predicate θ_p such that for any $c_1,\dots,c_k$, we have $\theta_p(c_1,\dots,c_k) = true$ iff $p(c_1,\dots,c_k) \in A_p$; for a (possibly non-ground) atom $\alpha = p(t_1,\dots,t_k)$, we write $\theta(\alpha)$ for $\theta_p(t_1,\dots,t_k)$. We can then build from a non-ground program Π an abstract non-ground program $omit(\Pi,A)$ according to the abstraction m_A, by mapping every rule $r : \alpha \leftarrow B$ in Π to a set $omit(r,A)$ of rules such that

$$omit(r,A) \text{ includes } \begin{cases} r & \text{if } A_{pred(\beta)} = \emptyset \text{ for all } \beta \in \{\alpha\} \cup B^{\pm}, \\ \alpha \leftarrow B, not\ \theta(\beta) & \text{if } A_{pred(\beta)} \neq \emptyset \wedge \beta \in \{\alpha\} \cup B^{\pm}, \\ \{\alpha\} \leftarrow B \setminus \{\beta\}, \theta(\beta) & \text{if } \beta \in B \wedge \alpha \neq \bot \wedge \theta(\beta) \text{ is satisfiable}, \\ \emptyset & \text{otherwise}, \end{cases}$$

and no other rules. The steps above assume that in a rule a most one predicate to omit occurs in a single atom β. However, the steps can be readily lifted to consider omitting a set $\{\beta_1,\dots,\beta_n\}$ of atoms with multiple predicates from the rules. For this, $\alpha \leftarrow B, not\ \theta(\beta)$ will be converted into $\alpha \leftarrow B, not\ \theta(\beta_1),\dots, not\ \theta(\beta_n)$ and $\{\alpha\} \leftarrow B \setminus \{\beta\}, \theta(\beta)$ gets converted into a set of rules $\{\alpha\} \leftarrow B \setminus \{\beta_1,\dots,\beta_n\}, \theta(\beta_1);\dots;\{\alpha\} \leftarrow B \setminus \{\beta_1,\dots,\beta_n\}, \theta(\beta_n)$.

Example 16 (Example 15 continued) *Say we want to omit $c(X)$ for $X{<}3$, i.e., $A = \{c(1),c(2)\} = A_c$. We have $\theta(c(1)) = \theta(c(2)) = true$ and $\theta(c(X)) = false$, for $X \in \{3,...,5\}$. The abstract non-ground program $omit(\Pi,A)$ is*

$$\begin{aligned} a(X_1,X_2) &\leftarrow c(X_1), b(X_2), not\ \theta(c(X_1)). \\ \{a(X_1,X_2)\} &\leftarrow b(X_2), \theta(c(X_1)). \\ d(X_1,X_2) &\leftarrow a(X_1,X_2), X_1 \leq X_2. \end{aligned}$$

The abstract answer sets with facts $b(2),\theta(c(1)),\theta(c(2))$ are $\{\{b(2)\},\{b(2),a(2,2),d(2,2)\},\ \{b(2),a(1,2),\ d(1,2)\},$ and $\{b(2),a(1,2),a(2,2),d(1,2),d(2,2)\}\}$. The program $omit(\Pi,A)$ is over-approximating Π while not introducing that many abstract answer sets as in the coarser abstraction in Example 15.

For determining bad omissions in non-ground programs, if lifting the current debugging rules is not scalable, other meta-programming ideas (Gebser et al., 2008; Oetsch et al., 2010) can be used. The issue that arises with the non-ground case is having lots of guesses to catch the inconsistency. Determining a reasonable set of bad omission atoms requires optimizations which makes solving the debugging problem more difficult.

7.2 Disjunctive Programs

For disjunctive programs, splitting the disjunctive rules yields an over-approximation.

Proposition 20 *For a program Π' constructed from a given Π by splitting rules of form $\alpha_{0_1} \vee \cdots \vee \alpha_{0_k} \leftarrow B(r)$ into $\{\alpha_{0_1}\} \leftarrow B(r); \ldots ; \{\alpha_{0_k}\} \leftarrow B(r)$, we have $AS(\Pi) \subseteq AS(\Pi')$.*

The current abstraction method can then be applied over Π'. However, it is possible that for an unsatisfiable Π the constructed Π' becomes satisfiable; the reason for unsatisfiability of Π can then not be grasped.

The approach from above can be extended to disjunctive programs Π, by injecting auxiliary atoms to disjunctive heads in order to cover the case where the body does not fire in the original program. To obtain with a given set A of atoms an abstract disjunctive program $omit(\Pi, A)$, we define abstraction of disjunctive rules $r : \alpha_1 \vee \cdots \vee \alpha_n \leftarrow B$ in Π, where $n \geq 2$ and all $\alpha_i \neq \bot$ are pairwise distinct, as follows.

$$omit(r, A) = \begin{cases} r & \text{if } A \cap B^{\pm} = \emptyset \wedge A \cap \{\alpha_1, \ldots, \alpha_n\} = \emptyset, \\ \alpha_1 \vee \cdots \vee \alpha_k \vee x \leftarrow m_A(B) & \text{if } A \cap \{\alpha_1, \ldots, \alpha_n\} = \{\alpha_{k+1}, \ldots, \alpha_n\} \wedge k \geq 1, \\ \alpha_1 \vee \cdots \vee \alpha_n \vee x \leftarrow m_A(B) & \text{if } A \cap B^{\pm} \neq \emptyset \wedge A \cap \{\alpha_1, \ldots, \alpha_n\} = \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$$

where x is a fresh auxiliary atom. Further development of the approach for disjunctive programs in a syntax preserving manner remains as future work.

7.3 Further Solution Aspects

The abstraction approach that we presented is focused on the syntactic level of programs, and it aims to preserve the structure of the given program. Thus, depending on the particular encoding that is used to solve a particular problem, the abstraction process may provide results that, from the semantic view of the problem, can be of quite different quality.

For illustration, consider a variant of the graph coloring encoding with a rule

$$colorUsed(Y) \leftarrow colored(X,Y), node(X), color(Y)$$

which records that a certain color is used in the coloring solution, and where $colorUsed(Y)$ is then used in other rules for further reasoning. Omitting nodes of the graph means omitting the ground atoms that involve with them; this will cause to have a choice rule $\{colorUsed(Y)\}$ for each color Y in the constructed abstract program. However, these guesses could immediately cause the occurrence of spurious answer sets due to the random guesses of *colorUsed*. Thus, one may need to add back many of the atoms in order to get rid of the spurious guesses.

Other aspects that apparently will have an influence on the quality of abstraction results is the way in which refinements are made and the choice of the initial abstraction. We considered possible strategies for this in order to help with the search, and we tested their effects in some of the benchmarks. The first strategy, described in Section 7.3.1, is on refining the reasoning step for determining bad omissions, while the the second, described in Section 7.3.2, is on making a more intuitive decision than a random choice for the initial set of omitted atoms.

7.3.1 Bad omission determination

It may happen that in a refinement step no put-back set is found that eliminates the spurious answer set. Therefore, we consider further reasoning for bad omission determination to see whether it can be useful in order to mitigate this behavior.

Example 17 *Consider the following program Π, with the single answer set $I = \{c,d,a,b\}$, and its abstraction $\widehat{\Pi}_{\overline{a,d}}$, with $AS(\widehat{\Pi}_{\overline{a,d}}) = \{\{c\},\{c,b\}\}$*

Π	$\widehat{\Pi}_{\overline{\{a,d\}}}$	$\widehat{\Pi}_{\overline{\{a\}}}$
$r1 : b \leftarrow d.$	$\{b\}.$	$b \leftarrow d.$
$r2 : d \leftarrow c, a.$		$\{d\} \leftarrow c.$
$r3 : a \leftarrow c.$		
$r4 : c.$	$c.$	$c.$

The abstract answer set $\hat{I} = \{c\}$ is spurious, as a corresponding answer set of Π must contain a by $r3$, d by $r2$ and b by $r1$, which is impossible. Adding to Π the query $Q_{\hat{I}}^{\overline{\{a,d\}}} = \{\bot \leftarrow not\ c.; \bot \leftarrow b.\}$ does not satisfy rule $r1$, which results in determining d as badomit since $r1$ should not remain as a choice rule. However, adding it back does not eliminate the answer set $\hat{I}$, since then $r2$ becomes a choice rule in $\widehat{\Pi}_{\overline{\{a\}}}$ causing again the occurrence of $\hat{I}$.

An additional reasoning over the omitted rules in determining bad omissions as below helps in deciding $\{a,d\}$ as badly omitted in one refinement step, and adding them back gets rid of the spurious answer set $\{c\}$.

Reasoning over omitted rules. We considered an additional *badomit* type to help with catching the cases when putting back one omitted atom does not eliminate the spurious answer set.

- If a rule was omitted due to a badly omitted atom, it has an omitted atom in the body, and the abstract rule was applicable, then an additional bad omission is inferred.

$$
\begin{aligned}
badomit(A2, type4) \leftarrow & omitted(R), head(R,A1), absAp(R),\\
& badomit(A1), omittedAtomFrom(A2,R).
\end{aligned}
$$

The idea is as follows: if some atom a, which was decided to be badly omitted, occurs in the head of a rule r, then once a is put back r will also be put back. However if $B(r)$ has some other omitted atom, then r will be put back as a choice rule. If this rule was also applicable in the abstract program for the given interpretation I, then once it has been put back as a choice rule, it will still be applicable for some $I' = I \cup \{a\}$ or $I'' = I$. Thus, the choice over $H(r)$ may again have the same spurious answer set determined.

Experiments. Figure 10 shows the conducted experiments with the additional bad omission detection. Observe that compared with the results in Figure 7, for the DS benchmarks the number of refinement steps and the time spent decreased since more omitted atoms were decided to be badly omitted in one step. Also we can see that the final set A_{final} of omitted atoms remains larger with the heuristics. On the other hand, this heuristic does not have a positive effect on the quality of the obtained minimal blockers. However, the results for the AA benchmarks are different. Although a larger final set of omitted atoms A_{final} were

Figure 10: Heuristic over *badomit* detection. The three entries in a cell, e.g., 0.41 / 0.51 / 0.63 in cell (DS, $\frac{|A_{final}|}{|\mathscr{A}|}$), are for 50% / 75% / 100% initial omission.

Π	$\frac{\lvert A_{final}\rvert}{\lvert\mathscr{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathscr{A}\rvert}$	t (sec)
DS	0.41	1.57	2.96	0.11	1.51
	0.51	3.03	5.06	0.10	1.00
	0.63	4.45	7.12	0.09	0.55
AA	0.11	5.02	8.91	0.37	9.66
	0.13	6.91	12.38	0.36	9.14
	0.15	8.11	14.27	0.35	8.86

computed for A_{init} with 100% (15% instead of 0.01% in Figure 7), the overall time spent and the refinement steps for obtaining some A_{final} increased. On the other hand, smaller minimal blockers were computed.

The results show that the considered strategy does not obtain the expected results on every program, as the structure of the programs matters.

7.3.2 Initial omission set selection

A possible strategy for setting up the initial omission set is to look at the occurrences of atoms in rule bodies and to select atoms that occur least often, as intuitively, atoms that occur less in the rules should be less relevant with the unsatisfiability.

Experiments In Figure 11 we see the results of choosing as initial omission 50% and 75% of the objects in increasing order by number of their occurrences. In the benchmarks GC^3, when omitting 75% of the least occurring nodes, two of the instances hit timeout during the Clingo call when searching for an optimal number of *badomit* atoms, and one instance hit timeout when computing some A_{final}, again spending most of the time in Clingo calls. The time increase for finding some optimized number of *badomit* atoms is due to many possible *badomit* atoms among the omitted atoms in the particular instances.

An interesting observation is that omitting 75% of the least occurring nodes results in larger A_{final} sets: while random omission removes on average 31% of the atoms (Figure 7), with the strategy added it increases to 67%. This result matches the intuition behind the strategy: the nodes that are not involved in the reasoning should not really be the cause of non-colorability. We also observe a positive effect on the quality of the computed $\subseteq$-minimal blocker sets, which are smaller in size, only 15% of the atoms for 50% and 75% initial omission, while before they were 16% and 17% (Figure 7), respectively.

For the AA benchmarks, compared to Figure 7 the strategy made it possible to obtain larger A_{final} sets. However, overall it does not show a considerable effect on the number of refinement steps or on the quality of the computed $\subseteq$-minimal blocker sets as in GC^3. We additionally performed experiments with full minimization of *badomit#* in the refinement step (Figure 12). Compared to the results in Figure 8, we can observe that larger A_{final} sets were obtained, and there were no timeouts when determining the *badomit* atoms in the refinement steps. The search for optimizing the number of *badomit* atoms is easier due to doing the search among the omitted atoms that have the least dependency.

For the DS benchmarks, although strategy reduced the average refinement steps and time, it had a negative effect on the quality of the $\subseteq$-minimal blocker sets as they are much larger (13% and 14% for initial

Figure 11: Heuristic over A_{init}. The two entries in a cell, e.g., 0.48 / 0.67 in cell (GC^3, $\frac{|A_{final}|}{|\mathscr{A}|}$), are for 50% / 75% initial omission.

Π	$\frac{\lvert A_{final}\rvert}{\lvert\mathscr{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathscr{A}\rvert}$	t (sec)
GC^3	0.48	0.26	1.42	0.15	1.33
	0.67	1.06	2.46 (3)	0.15	0.62
AA	0.22	3.22	6.69	0.37	8.25
	0.23	4.20	8.77	0.37	8.08
DS	0.35	0.38	1.66	0.13	2.46
	0.42	1.88	4.50	0.14	2.24

Figure 12: Heuristic over A_{init} with full minimization on *badomit*#. The two entries in a cell, e.g., 0.28 / 0.35 in cell (AA, $\frac{|A_{final}|}{|\mathscr{A}|}$), are for 50% / 75% initial omission.

Π	$\frac{\lvert A_{final}\rvert}{\lvert\mathscr{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathscr{A}\rvert}$	t (sec)
AA	0.28	7.49	14.29	0.35	7.62
	0.35	11.07	24.31	0.35	6.87

omission of 50% and 75% of tasks, instead of 10% and 9% as in Figure 7, respectively).

8 Related Work

Although abstraction is a well-known approach to reduce problem complexity in computer science and artificial intelligence, it has not been considered so far in ASP. In the context of logic programming, abstraction has been studied many years back in (Cousot and Cousot, 1992). However, the focus was on the use of abstract interpretations and termination analysis of programs, and moreover stable semantics was not addressed. In planning, abstraction has been used for different purposes; two main applications are plan refinement (Sacerdoti, 1974; Knoblock, 1994), which is concerned with using abstract plans computed in an abstract space to find a concrete plan, while abstraction-based heuristics (Edelkamp, 2001; Helmert et al., 2014) deals with using the costs of abstract solutions as a heuristic to guide the search for a plan. Pattern databases (Edelkamp, 2001) is a notion of abstraction which aims at projecting the state space to a set of variables, called a 'pattern'. In contrast, merge & shrink abstraction (Helmert et al., 2014) starts with a suite of single projections, and then computes a final abstraction by merging them and shrinking. In the sequel, we address related issues in the realm of ASP.

8.1 ASP Debugging

Investigating inconsistent ASP programs has been addressed in several works on debugging (Brain et al., 2007; Oetsch et al., 2010; Dodaro et al., 2015; Gebser et al., 2008), where the basic assumption is that one has an inconsistent program and an interpretation as expected answer set. In our case, we do not have a candidate solution but are interested in finding the minimal projection of the program that is inconsistent.

Through abstraction and refinement, we are obtaining candidate abstract answer sets to check in the original program. Importantly, the aim is not to debug the program itself, but to debug (and refine) the abstraction that has been constructed.

Different from other works, (Dodaro et al., 2015) computed the unsatisfiable cores (i.e., the set of atoms that, if true, causes inconsistency) for a set of assumption atoms and finds a diagnosis with it. The user is queried about the expected behavior, to narrow down the diagnosed set. In our work, such an interaction is not required and the set of blocker atoms that was found points to an abstract program (a projection of the original program) which shows all the rules (or projection of the rules) that are related with the inconsistency.

The work by (Syrjänen, 2006) is based on identifying the conflict sets that contain mutually incompatible constraints. However for large programs, the smallest input program where the error happens must be found manually. Another related work is (Pontelli et al., 2009), which gives justifications for the truth values of atoms with respect to an answer set by graph-based explanations that encode the reasons for these values. Notably, justifications can be computed offline or online when computing an answer set, where they may be utilized for program debugging purposes. The authors demonstrated how their approach can be used to guide the search for consistency restoring in CR-Prolog (Balduccini and Gelfond, 2003), by identifying restoral rules that are needed to resolve conflicts between literals detected from their justifications. However, the latter hinge on (possibly partial) interpretations, and thus do not provide a strong explanation of inconsistency as blockers, which are independent of particular interpretations.

8.2 Unsatisfiable Cores in ASP

A well-known notion for unsatisfiability are minimal unsatisfiable subsets (MUS), also known as *unsatisfiable cores* (Liffiton and Sakallah, 2008; Lynce and Silva, 2004). It is based on computing, given set of constraints respectively formulas, a minimal subset of the constraints that explains why the overall set is unsatisfiable. Unsatisfiable cores are helpful in speeding up automated reasoning, but have beyond many applications and a key role e.g. in model-based diagnosis (Reiter, 1987) and in consistent query answering (Arenas et al., 1999).

In ASP, unsatisfiable cores have been used in the context of computing optimal answer sets (Alviano and Dodaro, 2016; Andres et al., 2012), where for a given (satisfiable) program, weak constraints are turned into hard constraints; an unsatisfiable core of the modified program that consists of rewritten constraints allows one to derive an underestimate for the cost of an optimal answer set, since at least one of the constraints in the core can not be satisfied. However, if the original program is unsatisfiable, such cores are pointless. In the recent work (Alviano et al., 2018), unsatisfiable core computation has been used for implementing cautious reasoning. The idea is that modern ASP solvers allow one to search, given set of assumption literals, for an answer set. In case of failure, a subset of these literals is returned that is sufficient to cause the failure, which constitutes an unsatisfiable core. Cautious consequence of an atom amounts then to showing the the negated atom is an unsatisfiable core.

Intuitively, unsatisfiable cores are similar in nature to spurious abstract answer sets, since the latter likewise to not permit to complete a partial answer set to the whole alphabet. More formally, their relationship is as follows.

Technically, an *unsatisfiable (u-) core* for a program Π is an assignment I over a subset $C \subseteq \mathscr{A}$ of the atoms such that Π has no answer set J that is compatible with I, i.e., such that $J|_C = I$ holds. We then have the following property.

Proposition 21 *Suppose that $\hat{I} \in AS(omit(\Pi, A))$ for a program Π and a set A of atoms. If $\hat{I}$ is spurious, then*

$\hat{I}$ is a u-core of Π (w.r.t. $\mathscr{A} \setminus A$). Furthermore, if A is maximal, i.e., no $A' \supset A$ exists such that $omit(\Pi, A')$ has some (spurious) answer set $\widehat{I'}$ such that $\hat{I}|_{\overline{A'}} = \widehat{I'}$, then I is a minimal core.

That is, spurious answer sets are u-cores; however, the converse fails in that cores C are not necessarily spurious answer sets of the corresponding omission $A = \mathscr{A} \setminus \mathscr{A}(C)$, where $\mathscr{A}(C)$ are the atoms that occur in C. E.g., for the program with the single rule

$$r : a \leftarrow b, not\ a.$$

the set $C = \{b\}$ is a core, while C is not an answer set of $omit(\{r\}, \{a\})) = \emptyset$. Intuitively, the reason is that C lacks foundedness for the abstraction, as it assigns b true while there is no way to derive b from the rules of the program, and thus b must be false in every answer set. As C is a minimal u-core, the example shows that also minimal u-cores may not be spurious answer sets.

Thus, spurious answer sets are a more fine-grained notion of relative inconsistency than (minimal) u-cores, which accounts for a notion of weak satisfiability in terms of the abstracted program. In case of an unsatisfiable program Π, each blocker set C for Π naturally gives rise to u-cores in terms of arbitrary assignments I to the atoms in $\mathscr{A} \setminus C$; in this sense, blocker sets are conceptually a stronger notion of inconsistency explanation than u-cores, in which minimal blocker sets and minimal u-cores remain unrelated in general.

8.3 Forgetting

Forgetting is an important operation in knowledge representation and reasoning, which has been studied for many formalisms and is a helpful tool for a range of applications, cf. (Delgrande, 2017; Eiter and Kern-Isberner, 2018). The aim of forgetting is to reduce the signature of a knowledge base, by removing symbols from the formulas in it (while possibly adding new formulas) such that the information in the knowledge base, given by its semantics that may be defined in terms of models or a consequence relation, is invariant with respect to the remaining symbols; that is, the models resp. consequences for them should not change after forgetting.

Due to nonmonotonicity and minimality of models, forgetting in ASP turned out to be a nontrivial issue. It has been extensively studied in the form of introducing specific operators that follow different principles and obey different properties; we refer to (Gonçalves et al., 2017; Leite, 2017) for a survey and discussion. The main aim of forgetting in ASP as such is to remove/hide atoms from a given program, while preserving its semantics for the remaining atoms. As atoms in answer sets must be derivable, this requires to maintain dependency links between atoms. For example, forgetting the atom b from the program $\Pi = \{a \leftarrow b.;\ b \leftarrow c.\}$ is expected to result in a program Π' in which the link between a and c is preserved; this intuitively requires to have the rule $a \leftarrow c$ in Π'. The various properties that have been introduced as postulates or desired properties for an ASP forgetting operator mainly serve to ensure this outcome; forgetting in ASP is thus subject to more restricted conditions than abstraction.

Atom omission as we consider it is different from forgetting in ASP as it aims at a deliberate over-approximation of the original program that may not be faithful; furthermore, our omission does not resort to language extensions such as nested logic programs that might be necessary in order to exclude non-faithful abstraction; notably, in the ASP literature under-approximation of the answer sets was advocated if no language extensions should be made (Eiter and Wang, 2008).

Only more recently over-approximation has been considered as a possible property of forgetting in ASP in (Delgrande and Wang, 2015), which was later named *weakened Consequence (WC)* in (Gonçalves et al., 2016):

(WC) Let Π be a disjunctive logic program, let A be a set of atoms, and let X be an answer set for Π. Then $X \setminus A$ is an answer set for $\mathit{forget}(\Pi, A)$.

That is, $AS(\Pi)|_{\overline{A}} \subseteq AS(\mathit{forget}(\Pi, A))$ should hold. This property amounts to the notion of over-approximation that we achieve in Theorem 1. However, according to (Gonçalves et al., 2016), this property is in terms of proper forgetting only meaningful if it is combined with further axioms. Our results may thus serve as a base for obtaining such combinations; in turn, imposing further properties may allow us to prune spurious answer sets from the abstraction.

9 Conclusion

Abstraction is a well-known approach to reduce problem complexity by stepping to simpler, less detailed models or descriptions. In this article, we have considered this hitherto in Answer Set Programming neglected approach, and we have presented a novel method for abstracting ASP programs by omitting atoms from the rules of the programs. The resulting abstract program can be efficiently constructed and has rules similar to the original program and is a semantic over-approximation of the latter, i.e., each original answer set is covered by some abstract answer set. We have investigated semantic and computational properties of the abstraction method, and we have presented a refinement method for eliminating spurious answer sets by adding badly omitted atoms back. The latter are determined using an approach inspired from previous work on debugging ASP programs.

An abstraction and refinement approach like the one that we presented may be used for different purposes. We have demonstrated as a show case giving explanations of the unsatisfiability of ASP programs, which can be achieved in terms of particular sets of omitted atoms, called blockers, for which no truth assignment will lead to an answer set. Thanks to the structure-preserving nature of the abstraction method, this allows one to narrow down the focus of attention to the rules associated with the blockers. Experimental results collected with a prototype implementations have shown that, in this way, strong explanations for the cause of inconsistency can be found. They would not have been easily visible if we had applied a pure semantic approach in which connections between atoms might get lost by abstractions. We have briefly discussed how the approach may be extended to the non-ground case and to disjunctive programs, and we have addressed some further aspects that can help with the search.

Outlook and future work. There are several avenues of research in order to advance and complement this initial work on abstraction in ASP. Regarding over-approximation, the current abstraction method can be made more sophisticated in order to avoid introducing too many spurious answer sets. This, however, will require to conduct a more extensive program analysis, as well as to have non-modular program abstraction procedures which do not operate on a rule by rule basis; to what extent the program structure can be obtained, and understanding the trade-off between program similarity and answer set similarity are interesting research questions.

Another direction is building a highly efficient implementation. The current experimental prototype has been built on top of legacy code and tools such as Spock (Brain et al., 2007) from previous works; there is a lot of room for significant performance improvement. However, even for the current, non-optimized implementation it is already possible to see benefits in terms of qualitative improvements of the results. An optimized implementation may lead to view abstraction under a performance aspect, which then becomes part of a general ASP solving toolbox.

Yet another direction is to broaden the classes of programs to which abstraction can be fruitfully applied. We have briefly discussed non-ground and disjunctive programs, for which abstraction needs to be worked out, but also other language extensions such as aggregates, nested implication or program modules (which are naturally closes relatives to abstraction) are interesting topics. In particular, for non-ground programs other, natural forms of abstraction are feasible; e.g., to abstract over individuals of the domain of discourse, or predicate abstraction. The companion work (Saribatur and Eiter, 2018) studies the former issue.

References

ALVIANO, M., CALIMERI, F., CHARWAT, G., DAO-TRAN, M., DODARO, C., IANNI, G., KRENNWALLNER, T., KRONEGGER, M., OETSCH, J., PFANDLER, A., ET AL. 2013. The fourth answer set programming competition: Preliminary report. In *Proc. LPNMR*. Springer, 42–53.

ALVIANO, M. AND DODARO, C. 2016. Anytime answer set optimization via unsatisfiable core shrinking. *Theory and Practice of Logic Programming 16,* 5-6, 533–551.

ALVIANO, M., DODARO, C., JÄRVISALO, M., MARATEA, M., AND PREVITI, A. 2018. Cautious reasoning in asp via minimal models and unsatisfiable cores. *CoRR*. abs/1804.08480.

ANDRES, B., KAUFMANN, B., MATHEIS, O., AND SCHAUB, T. 2012. Unsatisfiability-based optimization in clasp. In *Proc. ICLP*. Vol. 17. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 211–221.

ARENAS, M., BERTOSSI, L. E., AND CHOMICKI, J. 1999. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, V. Vianu and C. H. Papadimitriou, Eds. ACM Press, 68–79.

BALDUCCINI, M. AND GELFOND, M. 2003. Logic programs with consistency-restoring rules. In *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*, J. McCarthy and M.-A. Williams, Eds. 9–18.

BANIHASHEMI, B., DE GIACOMO, G., AND LESPÉRANCE, Y. 2017. Abstraction in situation calculus action theories. In *Proc. of AAAI*. 1048–1055.

BRAIN, M., GEBSER, M., PÜHRER, J., SCHAUB, T., TOMPITS, H., AND WOLTRAN, S. 2007. Debugging asp programs by means of asp. In *Proc. LPNMR*. Springer, 31–43.

BRASS, S. AND DIX, J. 1997. Characterizations of the disjunctive stable semantics by partial evaluation. *J. Log. Program. 32,* 3, 207–228.

BREWKA, G., EITER, T., AND TRUSZCZYŃSKI, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

BUSS, S., KRAJÌČEK, J., AND TAKEUTI, G. 1993. On provably total functions in bounded arithmetic theories. In *Arithmetic, Proof Theory and Computational Complexity*, P. Clote and J. Krajìček, Eds. Oxford University Press, 116–61.

CERUTTI, F., GIACOMIN, M., AND VALLATI, M. 2016. Generating structured argumentation frameworks: AFBenchGen2. In *Proc. COMMA*, P. Baroni, T. F. Gordon, T. Scheffler, and M. Stede, Eds. Vol. 287. IOS Press, 467–468.

CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *JACM 50,* 5, 752–794.

CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. 1994. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1512–1542.

COUSOT, P. AND COUSOT, R. 1992. Abstract interpretation and application to logic programs. *The Journal of Logic Programming 13,* 2, 103 – 179.

CULBERSON, J. C. AND SCHAEFFER, J. 1998. Pattern databases. *Computational Intelligence 14,* 3, 318–334.

DELGRANDE, J. P. 2017. A knowledge level account of forgetting. *J. Artif. Intell. Res. 60*, 1165–1213.

DELGRANDE, J. P. AND WANG, K. 2015. A syntax-independent approach to forgetting in disjunctive logic programs. In *AAAI*. 1482–1488.

DODARO, C., GASTEIGER, P., MUSITSCH, B., RICCA, F., AND SHCHEKOTYKHIN, K. 2015. Interactive debugging of non-ground asp programs. In *Proc. LPNMR*. Springer, 279–293.

EDELKAMP, S. 2001. Planning with pattern databases. In *Proc. ECP*. 13–24.

EITER, T. AND KERN-ISBERNER, G. 2018. A brief survey on forgetting from a knowledge representation and reasoning perspective. *KI – Künstliche Intelligenz*. Online `http://link.springer.com/article/10.1007/s13218-018-0564-6`.

EITER, T., LEONE, N., MATEIS, C., PFEIFER, G., AND SCARCELLO, F. 1998. The KR system dlv: Progress report, comparisons and benchmarks. *Proc. KR 98*, 406–417.

EITER, T. AND WANG, K. 2008. Semantic forgetting in answer set programming. *Artif. Intell. 172,* 14, 1644–1672.

FABER, W., LEONE, N., AND PFEIFER, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proc. JELIA*. LNCS, vol. 3229. Springer, 200–212.

GAGGL, S. A., LINSBICHLER, T., MARATEA, M., AND WOLTRAN, S. 2016. Introducing the second international competition on computational models of argumentation. In *Proc.SAFA*. 4–9.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. 2008. Engineering an incremental ASP solver. In *Proc. ICLP*. 190–205.

GEBSER, M., KAUFMANN, B., KAMINSKI, R., OSTROWSKI, M., SCHAUB, T., AND SCHNEIDER, M. 2011. Potassco: The Potsdam Answer Set Solving Collection. *AI Comm. 24,* 2, 107–124.

GEBSER, M., PÜHRER, J., SCHAUB, T., AND TOMPITS, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proc. AAAI*. Vol. 8. 448–453.

GEISSER, F., KELLER, T., AND MATTMÜLLER, R. 2016. Abstractions for planning with state-dependent action costs. In *ICAPS*. 140–148.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3, 365–385.

GIUNCHIGLIA, F. AND WALSH, T. 1992. A theory of abstraction. *AIJ 57,* 2-3, 323–389.

GONÇALVES, R., KNORR, M., AND LEITE, J. 2016. The ultimate guide to forgetting in answer set programming. In *KR*. 135–144.

GONÇALVES, R., KNORR, M., LEITE, J., AND WOLTRAN, S. 2017. When you must forget: Beyond strong persistence when forgetting in answer set programming. *TPLP 17,* 5-6, 837–854.

HELMERT, M., HASLUM, P., HOFFMANN, J., AND NISSIM, R. 2014. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM 61,* 3, 16.

JANHUNEN, T., NIEMELÄ, I., SEIPEL, D., SIMONS, P., AND YOU, J.-H. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM TOCL 7,* 1 (Jan.), 1–37.

JANOTA, M. AND MARQUES-SILVA, J. 2016. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell. 233*, 73–83.

KNOBLOCK, C. A. 1994. Automatically generating abstractions for planning. *Artificial Intelligence 68,* 2, 243–302.

KOUVAROS, P. AND LOMUSCIO, A. 2015. A counter abstraction technique for the verification of robot swarms. In *Proc. of AAAI.*

LEITE, J. 2017. A bird's-eye view of forgetting in answer-set programming. In *Proc. LPNMR*, M. Balduccini and T. Janhunen, Eds. LNCS, vol. 10377. Springer, 10–22.

LIFFITON, M. H. AND SAKALLAH, K. A. 2008. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning 40,* 1, 1–33.

LIFSCHITZ, V., TANG, L. R., AND TURNER, H. 1999. Nested expressions in logic programs. *Ann. Math. Artif. Intell. 25,* 3-4, 369–389.

LYNCE, I. AND SILVA, J. P. M. 2004. On computing minimum unsatisfiable cores. In *Proc. SAT.*

OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2010. Catching the ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Prog. 10,* 4-6, 513–529.

PEARCE, D. 2004. Simplifying logic programs under answer set semantics. In *Logic Programming*, B. Demoen and V. Lifschitz, Eds. 210–224.

PONTELLI, E., SON, T. C., AND ELKHATIB, O. 2009. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming 9,* 1, 1–56.

REITER, R. 1987. A theory of diagnosis from first principles. *Artif. Intell. 32,* 1, 57–95.

SACERDOTI, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence 5,* 2, 115–135.

SARIBATUR, Z. G. AND EITER, T. 2018. Towards abstraction in ASP with an application on reasoning about agent policies. *CoRR abs/1809.06638*. In: Informal Proc. 11th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'18@FLoC).

SYRJÄNEN, T. 2006. Debugging inconsistent answer set programs. In *Proc. NMR*. Vol. 6. 77–83.

VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM (JACM) 38,* 3, 619–649.

WATTS, D. J. AND STROGATZ, S. H. 1998. Collective dynamics of “small-world" networks. *Nature 393*, 440–442.