Omission-based Abstraction for Answer Set Programs

Zeynep G. Saribatur, Thomas Eiter
Institute of Logic and Computation, TU Wien, Austria

Abstract

Abstraction is a well-known approach to reduce program complexity by over-approximating the problem with a deliberate loss of information. It has not been considered so far in the context of Answer Set Programming, a convenient tool for problem solving. In this paper, we introduce a method to automatically abstract ground ASP programs that preserves their structure by reducing the vocabulary. Such an abstraction makes it possible to generate partial answer set candidates, which can help with the approximation of reasoning. Faithful (non-spurious) abstractions may be used to represent the projection of answer sets and to guide solvers in answer set construction. In order to deal with the unavoidably introduced spurious answer sets, we employ an ASP debugging approach to help with the refinement of the abstraction. We investigate the usage of such an abstraction to obtain explanations of unsatisfiable programs as a show case.

Introduction

Abstraction is a widely used approach in computing solutions for hard problems by over-approximating them. By a deliberate loss of information, the problem is approximated to achieve a smaller or simpler state space, at the price of spurious counterexamples to the behavior. The well-known counterexample guided abstraction refinement (CEGAR) (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.

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 search size, 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 (Brain et al. 2007; Oetsch, Pührer, and Tompits 2010; Dodaro et al. 2015; Gebser et al. 2008), which is based on providing the reason (i.e., occurring violations) on why an expected solution provided by the user does not exist. However, these methods do not address the question of why the program does not give any solutions. We approach the unsatisfiability of an ASP program differently with an interest in obtaining a projection of the program which shows the cause of the unsatisfiability, without an initial idea on expected solutions. The well-known notion of minimal unsatisfiable subsets (*unsatisfiable cores*) (Liffiton and Sakallah 2008; Lynce and Silva 2004) has also been used in the ASP context (Alviano and Dodaro 2016; Andres et al. 2012) (for further discussion see Related Work).

Our contributions are briefly summarized as follows.

- 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. In this paper, we focus on the ground case.
- 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 set* as a set of atoms such that abstraction to it preserves unsatisfiability of a program. A minimal blocker set then gives a projection of the program to the minimal cause of unsatisfiability.
- We derive complexity results for the notions, such as for checking for spurious answer sets, finding minimal sets of

atoms to put back in the refinement to eliminate a spurious solution, and computing a minimal blocker for a program.

- We report about preliminary experiments focusing on unsatisfiable programs and investigate computing minimal blockers of programs. We compare the results of the abstraction and refinement approach (*bottom-up*) with a naive *top-down* approach and observe that abstraction 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.

Preliminaries

A logic program Π is a set of rules r of the form

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

where each α_i is an atom 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 $\alpha_0 \leftarrow B^+(r), \mathit{not}\ B^-(r)$, where $B^+(r)$ (positive body) is the set $\{\alpha_1, \ldots, \alpha_m\}$ and $B^-(r)$ (negative body) the set $\{\alpha_{m+1}, \ldots, \alpha_n\}$, or $\alpha_0 \leftarrow B(r)$ where $B(r) = B^+(r) \cup B^-(r)$. We sometimes write B instead of $B(r)$ when talking about a particular rule. To group the rules with the same head α, we use $\mathit{def}(\alpha, \Pi) = \{r \in \Pi \mid H(r) = \alpha\}$. Rules with variables stand for the set of their ground instances. The set of ground atoms of Π is denoted by $\mathcal{A}$. Semantically, Π induces a set of answer sets (Gelfond and Lifschitz 1991), which are Herbrand models (sets I of ground atoms) of Π justified by the rules, in that I is a minimal model of $f\Pi^I = \{r \in \Pi \mid I \models B(r)\}$ (Faber, Leone, and Pfeifer 2004). The set of answer sets of a program Π is denoted as $AS(\Pi)$. A program Π is *unsatisfiable*, if $AS(\Pi) = \emptyset$. Common syntactic extensions are *choice rules* of the form $\{\alpha\} \leftarrow B$, which stands for the rules $\alpha \leftarrow B, \mathit{not}\ \alpha'$ and $\alpha' \leftarrow B, \mathit{not}\, \alpha$, where α' is a new atom, and cardinality constraints and conditional atoms (Simons, Niemelä, and Soininen 2002); in particular, $i_\ell\{\, \alpha_1(X) : \alpha_2(X)\,\}i_u$ is true whenever at least i_ℓ and at most i_u instances of $\alpha_1(X)$ subject to $\alpha_2(X)$ are true.

An interpretation I *falsifies* a rule r, if $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$, but $H(r) \notin I$. An atom α is *unsupported* by an interpretation I if for each $r \in \mathit{def}(\alpha, \Pi)$, $B^+(r) \not\subseteq I$ or $B^-(r) \cap I \neq \emptyset$. A set $A \subseteq \mathcal{A}$ 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. The *dependency graph* of Π, denoted G_Π, has vertices $\mathcal{A}$, (positive) edges from any $\alpha_0 \in H(r)$ to any $\alpha_1 \in B^+(r)$ and (negative) edges from any $\alpha_0 \in H(r)$ to any $\alpha_2 \in B^-(r)$, for all $r \in \Pi$. An *odd loop* means that an atom $\alpha \in \mathcal{A}$ 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.

Abstraction by Omission

Our aim is to over-approximate a given program through constructing a simpler program by reducing the vocabulary and preserving the behavior of the original program (i.e., the results of reasoning on the original program are not lost), at the cost of obtaining spurious solutions.

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

We refer to m as an *abstraction mapping*. In this paper, we focus on an omission-based abstraction.

Definition 2. Given a set $A \subseteq \mathcal{A}$, an *omission (abstraction) mapping* is $m_A : \mathcal{A} \to \mathcal{A} \cup \{\top\}$ 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*. We denote by $\mathit{omit}(\Pi, A)$ an abstraction of Π with an omission mapping that omits a set A of atoms.

Next we show a systematic way of building an omission-based abstraction of a given ASP program. When constructing an abstract program for a given mapping, the aim is to ensure that every original answer set I is mapped to some abstract answer set, while (unavoidably) some spurious abstract answer sets may be introduced. Thus, an over-approximation of the original program is achieved.

Our 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. In case of omitting non-ground atoms, all occurrences of the predicate will be omitted and introducing domain variables may be needed.

Program abstraction. We build from Π an abstract program $\mathit{omit}(\Pi, A)$ according to the abstraction m_A. For every rule $r : \alpha \leftarrow B$ in Π, if $\alpha \in A$, then $m_A(r) = \emptyset$; otherwise, for $\alpha \in (\mathcal{A} \setminus A) \cup \{\bot\}$ we have

$$m_A(r) = \begin{cases} r & \text{if} \quad A \cap B = \emptyset, \\ \{\alpha\} \leftarrow m_A(B) & \text{if} \quad A \cap B \neq \emptyset \wedge \alpha \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases}$$

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

Note that we treat default negated atoms, B^-, similarly, i.e., if some $\alpha \in B^- \cap A$, then we omit $\mathit{not}\ \alpha$ from B.

Example 1. Consider a program Π and its abstraction $\widehat{\Pi}_{\overline{A}}$ for $A = \{b, d\}$, according to the above steps.

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

We have $AS(\Pi) = \{\{c, a\}, \{d, b\}\}$ and $AS(\widehat{\Pi}_{\overline{A}}) = \{\{\}, \{c\}, \{c, a\}\}$. 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., $AS(\Pi)|_{\overline{A}} \subseteq AS(\widehat{\Pi}_{\overline{A}})$.

Notice that in $\widehat{\Pi}_{\overline{A}}$, constraints are omitted if the body contains an omitted atom. If instead the constraint gets shrunk by just omitting the atom from the body, then for some interpretation $\widehat{I}$, the body may be satisfied, causing $\widehat{I} \notin AS(\widehat{\Pi}_{\overline{A}})$, while this was not the case in Π for any $I \in AS(\Pi)$ with $I|_{\overline{A}} = \widehat{I}$. 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.

If in a rule r, the omitted non-ground atom $p(V_1, \ldots, V_n)$ in 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 2. Consider the following simple program Π:

$$a(X_1, X_2) \leftarrow c(X_1), b(X_2). \quad (1)$$
$$d(X_1, X_2) \leftarrow a(X_1, X_2), X_1 \leq X_2. \quad (2)$$

In omitting $c(X)$, while rule (2) remains the same, rule (1) changes to $0\{a(X_1, X_2) : dom(X_1)\}1 \leftarrow b(X_2)$. 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 answer sets with fact $b(2)$ become $\{b(2), a(1,2), d(1,2)\}$ and $\{b(2), a(2,2), d(2,2)\}$, which cover the original answers, i.e., each original answer set can be mapped to some abstract one.

For a semantical more fine-grained removal, e.g., removing $c(X)$ for $X<3$, rules may be split in cases, e.g., (1) into $X_1<3$ and $X_1 \geq 3$, and treated after renaming separately.

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 $A \subseteq \mathcal{A}$, it holds that $I|_{\overline{A}} \in AS(omit(\Pi, A))$.*

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.

Definition 3. An answer set $I|_{\overline{A}} \in AS(omit(\Pi, A))$ is *concrete* if $I \in AS(\Pi)|_{\overline{A}}$, and *spurious* otherwise.

In other words, a spurious abstract answer set can not be completed to an original answer set, i.e., for any $X \subseteq A$, $I = I|_{\overline{A}} \cup X \notin AS(\Pi)$. Let $Q^{\overline{A}}_{\widehat{I}}$ denote the query for an answer set that matches $\widehat{I}$, i.e., $Q^{\overline{A}}_{\widehat{I}} = \{\bot \leftarrow not\ \alpha \mid \alpha \in \widehat{I}\} \cup \{\bot \leftarrow \alpha \mid \alpha \in \overline{A} \setminus \widehat{I}\}$. As an alternative definition, $\widehat{I}$ is spurious iff $\Pi \cup Q^{\overline{A}}_{\widehat{I}}$ is unsatisfiable.

Upon encountering spurious answer sets, *refinement* of the abstraction is necessary by adding back some of the omitted atoms. Next, we define a notion to talk about the set of omitted atoms that need to be added back in order to get rid of a spurious answer set. The notation $\widehat{I}$ is sometimes replaced by I, when the abstraction is clear from the context.

Definition 4. For a spurious $I \in AS(omit(\Pi, A))$, a *put-back set* $PB_I \subseteq A$ is a set of atoms such that for $A' = A \setminus PB_I$, $\nexists J \in AS(omit(\Pi, A'))$ such that $J|_{\overline{A}} = I$.

After adding back the put-back atoms in the abstraction, the spurious answer set I is *eliminated* in the updated abstract program. Notice that $PB_I = A$ also holds, as putting all the atoms back would eliminate the spurious answer set.

The following properties can be easily seen.

Proposition 2. *For any program Π,*

(i) $omit(\Pi, \emptyset) = \Pi$ *and* $omit(\Pi, \mathcal{A}) = \emptyset$.

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

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

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

Omitting atoms in a program means projecting away those atoms in the concrete answer sets of a program.

Proposition 3. *If I is a concrete answer set of $omit(\Pi, A)$, then for every $A' \subseteq A$ some answer set I' of $omit(\Pi, A')$ exists such that $I'|_{\overline{A}} = I$.*

The next property is on convexity of spurious answer sets.

Proposition 4. *Let $I \in AS(omit(\Pi, A))$ be spurious and $A' \subseteq A \subseteq \mathcal{A}$. If some $I' \in AS(omit(\Pi, A'))$ exists s.t. $I'|_{\overline{A}} = I$ (i.e., I' is spurious), then for every A'' s.t. $A' \subseteq A'' \subseteq A$, $I'|_{\overline{A''}}$ is a spurious answer set of $omit(\Pi, A'')$.*

The next proposition shows that once a spurious answer set is eliminated by adding back some of the omitted atoms, this answer set will not show up again when further omitted atoms are added back.

Proposition 5. *Let $I \in AS(omit(\Pi, A))$ be spurious, and let $A' = A \setminus PB_I$ be some refinement where I is eliminated. For all $A'' \subseteq A'$, $\nexists I'' \in AS(omit(\Pi, A' \setminus A''))$ s.t. $I''|_{\overline{A}} = I$.*

Proof. Assume such an A'' exists. This means that, after eliminating I by adding back PB_I, by adding further atoms of A'' back, an interpretation $I'' \in AS(omit(\Pi, A' \setminus A''))$ that can be projected to I, i.e., $I''|_{\overline{A}} = I$, is obtained. However, since $omit(\Pi, A')$ is an over-approximation of $omit(\Pi, A' \setminus A'')$, by definition, some $I' \in AS(omit(\Pi, A'))$ exists with $I''|_{\overline{A'}} = I'$, a contradiction to the assumption that PB_I is a put-back set, i.e., $\nexists I' \in AS(omit(\Pi, A'))$ s.t. $I'|_{\overline{A}} = I$. □

Catching Unsatisfiability Reasons of Programs

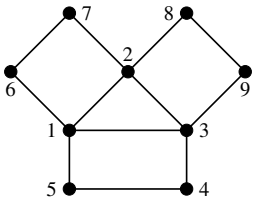
For an unsatisfiable program Π, by omitting the atoms and over-approximating the program one can obtain an abstract program which has some abstract answer set. Any such answer set will be spurious.

By Proposition 2-(iii), we can use omission abstraction and refinement to catch the true cause of inconsistency in a program. For this we introduce the following notions.

Definition 5. A set $C \subseteq \mathcal{A}$ of atoms is an *(answer set) blocker set* of Π, if $AS(omit(\Pi, \mathcal{A} \setminus C)) = \emptyset$.

In other words, when we keep the set C of atoms and omit the rest from Π to achieve the abstract program Π', the latter is still unsatisfiable. This means the atoms in C are blocking the occurrence of answer sets. No answer sets are possible as long as these atoms are present in the program. Notice that $C = \mathcal{A}$, i.e., no atom is omitted, is also a blocker set, while $C = \emptyset$, i.e., all atoms are omitted, is not such a set since $AS(omit(\Pi, \mathcal{A})) = \{\emptyset\}$.

Figure 1: A non 2-colorable graph



Definition 6. A blocker set $C \subseteq \mathcal{A}$ is $\subset$-*minimal*, if for all $C' \subset C$, $AS(omit(\Pi, \mathcal{A} \setminus C')) \neq \emptyset$.

For a minimal blocker set C, there is a *maximal unsatisfiable abstraction* $\mathcal{A} \setminus C$, which is the maximal set of atoms to omit while keeping the unsatisfiability of Π.

Omitting the atoms is about modifying or omitting the relevant ground rules from Π. The minimality focus can also be on the rules of the program. A set of (ground) rules $R = \{r_1, \ldots, r_n\}$ is the *minimal blocker rule set* if for any rule $r_i \in R$, $\Pi \setminus r_i$ is satisfiable.

Example 3 (Graph coloring)**.** Consider the graph coloring problem for the graph shown in Figure 1, which is not 2-colorable, due to the clique $1-2-3$. A common encoding for this problem is grounded to the given instance to obtain the below rules for all $N \in \{1, \ldots, 9\}$, and $C, C_1, C_2 \in \{1, 2\}$:

$$\{chosenColor(N, C)\}.$$
$$colored(N) \leftarrow chosenColor(N, C). \quad (3)$$
$$\leftarrow not\ colored(N).$$
$$\leftarrow chosenColor(N, C_1), chosenColor(N, C_2), C_1 \neq C_2.$$

In additional, for all nodes N_1, N_2 that contain an edge and colors $C \in \{1, 2\}$ there are rules of form

$$\leftarrow chosenColor(N_1, C), chosenColor(N_2, C).$$

Omitting a node in the graph corresponds to omitting the ground atoms related to the node. Omitting all the nodes except $\{1, 2, 3\}$ gives the minimal blocker rule set that consists of the ground rules (3) for $N \in \{1, \ldots, 3\}$ and the following constraints for $C \in \{1, 2\}, N_1, N_2 \in \{1, \ldots, 3\}$.

$$\leftarrow chosenColor(N_1, C), chosenColor(N_2, C), N_1 \neq N_2.$$

This abstract program is unsatisfiable and omitting further atoms in the abstraction gives spurious satisfiability.

Faithful Abstractions

Ideally, abstraction does not change the semantics of a program. Our next notion is to describe such abstractions.

Definition 7. 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., $AS(omit(\Pi, A)) = AS(\Pi)|_{\overline{A}}$.

Definition 8. A faithful abstraction $omit(\Pi, A)$ is *refinement-safe*, if for all $A' \subseteq A$, $omit(\Pi, A')$ has no spurious answer sets.

Notice that if Π is satisfiable, then $A = \mathcal{A}$ is a faithful abstraction, but it is often not refinement-safe, as with atoms added back choice rules may occur which can cause spurious answer sets.

Computational Complexity

In this section, we consider the complexity of some reasoning tasks associated with abstraction.

We note the following simple lemma and the proposition.

Lemma 6. *Given Π and A, (i) the program $omit(\Pi, A)$ is constructible in polynomial time, and (ii) checking whether $I \in AS(omit(\Pi, A))$ holds for a given I is feasible in polynomial time.*

Proposition 7. *Given a (ground) 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 Π w.r.t. A is* $\mathbf{NP}$*-complete (resp.* co$\mathbf{NP}$*-complete).*

Indeed, by the lemma we can check nondeterministically in polynomial time that $I|_{\overline{A}}$ is an answer set and that some answer set I' of Π exists such that $I'|_{\overline{A}} = I|_{\overline{A}}$; the $\mathbf{NP}$-hardness is immediate from the above proposition and the $\mathbf{NP}$-completeness of answer set existence.

Theorem 8. *Given a (ground) program Π and a set of atoms A, deciding whether some spurious answer set of Π w.r.t. A exists is Σ_2^p-complete.*

Proof (sketch). Some answer set I of $omit(\Pi, A)$ can be guessed and checked in polynomial time, and with the help of an $\mathbf{NP}$ oracle we can check whether I is spurious. The Σ_2^p-hardness can be shown by a reduction from evaluating a QBF $\exists X \forall Y\ E(X, Y)$, where $E(X, Y) = \bigvee_i D_i$ is a DNF

Construct a program Π as follows;

$$x_i \leftarrow not\ \overline{x_i}. \quad (4)$$
$$\overline{x_i} \leftarrow not\ x_i. \quad \text{for all } x_i \in X \quad (5)$$
$$y_j \leftarrow \overline{y_j}, not\ sat. \quad (6)$$
$$\overline{y_j} \leftarrow y_j, not\ sat. \quad \text{for all } y_j \in Y \quad (7)$$
$$sat \leftarrow l^*_{i_1}, \ldots l^*_{i_{n_i}}. \quad (8)$$

where $D_i = l_{i_1} \wedge \cdots \wedge l_{i_{n_i}}$ and l^* is as follows: $(l)^* = \overline{x_i}$, if $l = \neg x_i$; $= x_i$ if $l = x_i$; $= y_j$ if $l = y_j$; and $\overline{y_j}$ if $l = \neg y_i$.

For $A = Y \cup \overline{Y} \cup \{sat\}$, the answer sets I of $omit(\Pi, A)$ correspond 1-1 to truth assignments σ of X; any such $I = I_\sigma$ is spurious iff $E(\sigma(X), Y)$ is not a tautology. □

Recall that we call an abstraction $omit(\Pi, A)$ faithful, if $omit(\Pi, A)$ has no spurious answer sets.

Corollary 9. *Deciding, given a program Π and atoms $A \subseteq \mathcal{A}$, whether $omit(\Pi, A)$ is a faithful abstraction of Π is Π_2^p-complete.*

Next let us consider the computation of put-back sets. We recall that $\mathbf{FP}^{\mathbf{NP}}$ are the search problems for which a solution can be computed in polynomial time with an $\mathbf{NP}$ oracle, and $\mathbf{FP}^{\mathbf{NP}}$ is the same but under the restriction that all oracle calls have to be made at once. The class $\mathbf{FP}^{\Sigma_k^p}[log, wit]$, for $k \geq 1$, contains all search problems that can be solved in polynomial time with a witness oracle for Σ_k^p (Buss, Krajìček, and Takeuti 1993); a *witness* oracle for Σ_k^p returns in case of a yes-answer to a instance a polynomial size witness string that can be checked with an Σ_{k-1}^p oracle in polynomial time. In particular, for $k = 1$, i.e., for

$\mathbf{FP}^{\mathbf{NP}}[log, wit]$, one can use a SAT oracle and the witness is a satisfying assignment to a given SAT instance, cf. (Janota and Marques-Silva 2016).

Theorem 10. *Given a (ground) program Π, a set of atoms A, and a spurious answer set I of Π w.r.t. A, computing some (i) $\subseteq$-minimal resp. (ii) smallest size put-back set S for I is in case (i) feasible in $\mathbf{FP}^{\mathbf{NP}}$ and $\mathbf{FP}_{\parallel}^{\mathbf{NP}}$-hard resp. (ii) is $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-complete.*

Note that few $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-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 (Sketch). As for (i), we can compute such a set S by an elimination procedure: starting with $A' = \emptyset$, we repeatedly pick some atom $\alpha \in A \setminus A'$ and test (+) whether for $A'' = A' \cup \{\alpha\}$, the program $omit(\Pi, A'')$ has no answer set I'' such that $I''|_{\overline{A}} = I$; if yes, we set $A' := A''$ and make the next pick from A'. Upon termination, $S = A \setminus A'$ is a minimal put-back set. The test (+) can be done with an $\mathbf{NP}$ oracle. The hardness for $\mathbf{FP}_{\parallel}^{\mathbf{NP}}$ is shown by a reduction from computing, given programs $P_1, \ldots, P_n$ the answers $q_1, \ldots, q_n$ to whether P_i has some answer set.

The membership in case (ii) can be established by a binary search over put-back sets of bounded size using a Σ_2^p witness oracle. The $\mathbf{FP}^{\Sigma_2^P}[log, wit]$ hardness is shown by a reduction from the following problem: given a QBF $\Phi = \forall Y E(X, Y)$, compute a smallest size assignment σ to X such that $\forall Y E(\sigma(X), Y)$ evaluates to true, knowing that some σ exists, where the size of σ is the number of atoms set to true. The core idea is similar to the one in the proof of Theorem 8, but the construction is much more involved and needs significant modifications and extensions. □

Theorem 11. *Given a set $A_0 \subseteq \mathcal{A}$, computing a (i) $\subseteq$-maximal set A resp. (ii) largest size set A where $A \subseteq \mathcal{A} \setminus A_0$ such that $omit(\Pi, A)$ is a refinement-safe faithful abstraction is in case (i) in $\mathbf{FP}^{\mathbf{NP}}$ and $\mathbf{FP}_{\parallel}^{\mathbf{NP}}$-hard resp. (ii) $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-complete.*

Proof (sketch). (i) One sets $A := \emptyset$ and $S := \mathcal{A} \setminus A_0$ in the beginning 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 of Π.

Notably, the omission test (*) can be done with an $\mathbf{NP}$ oracle: the test fails iff for some A', the program $omit(\Pi, A' \cup \alpha)$ has a spurious answer set I'. In principle, the spurious check for I' is difficult (a co$\mathbf{NP}$-complete problem, by our results), but we can take advantage of knowing at this point that the abstraction $omit(\Pi, A')$ is faithful: so we only need to check whether I' is extendable to an answer set of $omit(\Pi, A')$, and not of Π itself; i.e., we only need to check that neither I' nor $I' \cup \alpha$ is an answer set of $omit(\Pi, A')$.

(ii) The of $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-completeness is similar as above for Theorem 10. In particular, the minimal put-back set C computed in the proof of the hardness part there is such that $\overline{C}$ is a minimal blocker. □

We remark that without refinement safety, the problem is likely to be more complex: deciding whether an abstraction is faithful is Π_2^p-complete and is trivially reducible to this problem.

Corollary 12. *Computing a (i) $\subseteq$-minimal resp. (ii) smallest size blocker $C \subseteq \mathcal{A}$ for a given program Π is (i) in $\mathbf{FP}^{\mathbf{NP}}$ and $\mathbf{FP}_{\parallel}^{\mathbf{NP}}$-hard resp. (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.

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 $\widehat{I}$, referred as *inconsistency of* Π *w.r.t.* $\widehat{I}$.

We use a meta-level debugging language (Brain et al. 2007) which is based on a tagging technique that allows one to control the formation 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, so 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.

Debugging Meta-program

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

Definition 9. Given Π, the program $\mathcal{T}_{meta}[\Pi]$ consists of the following rules for $r \in \Pi, \alpha_1 \in B^+(r), \alpha_2 \in B^-(r)$:

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

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 10. Given Π over $\mathcal{A}$, the following additional meta-programs are constructed:

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

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

2. $\mathcal{T}_C[\Pi, \mathcal{A}]$: for all $\alpha \in \mathcal{A} \backslash A$, with $def(\alpha, \Pi) = \{r_1, ..., r_k\}$:

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

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

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

In $\mathcal{T}_C[\Pi, \mathcal{A}]$, we do not guess over the atoms A if the 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., not 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.

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 $\widehat{I}$, i.e., $\Pi \cup Q^{\overline{A}}_{\widehat{I}}$. We reason about the inconsistency by inspecting the reason for having $\widehat{I} \in AS(omit(\Pi, A))$ due to some modified rules.

Definition 11. 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 $\widehat{I} \in AS(omit(\Pi, A))$

(i) a *spurious choice*, if $\widehat{I} \models m_A(B)$ and $\widehat{I} \models \overline{\alpha}$, but some model I of $\Pi \setminus r$ exists s.t. $I|_{\overline{A}} = \widehat{I}$ and $I \models B$.

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

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

Definition 12. Let $r : \alpha \leftarrow B$ be a rule in Π such that $B \cap A \neq \emptyset$. The abstract rule $m_A(r)$ introduces a *spurious loop-behavior* w.r.t. $\widehat{I}$, if some model I of Π exists s.t. $I|_{\overline{A}} = \widehat{I}$ and $I \models r$, but α is involved in a loop that is unfounded or is odd, due to some $\alpha' \in A \cap B$.

We show later in an example the need for reasoning on the two possible faulty loop behaviors.

Definition 13. An atom $\alpha \in A$ is a *bad-omission* w.r.t. a spurious answer set $\widehat{I}$ of $omit(\Pi, A)$, if some rule $r \in \Pi$ with $\alpha \in B(r)$ exists s.t. $\hat{r}$ introduces either a spurious choice, a spurious support or a spurious loop-behavior w.r.t. $\widehat{I}$.

Intuitively, for case (i) of Defn. 11, 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 $\widehat{I} \models B(\hat{r})$, 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 the issues arise due to omitting a constraint in the abstraction.

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

Definition 14. The bad omission determining program $\mathcal{T}_{badomit}$ is constructed using the abnormality atoms obtained from $\mathcal{T}_P[\Pi]$, $\mathcal{T}_C[\Pi, \mathcal{A}]$ and $\mathcal{T}_A[\mathcal{A}]$ 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:

$$badomit(X, type2) \leftarrow bl(R), head(R, H), ab_c(H), absAp(R), changed(R), omittedAtomFrom(X, R).$$

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

$$faulty(X) \leftarrow ab_l(X), inOddLoop(X, X1), omittedAtom(X1).$$
$$faulty(X) \leftarrow ab_l(X), inPosLoop(X, X1), omittedAtom(X1).$$
$$badomit(X1, type3) \leftarrow faulty(X), head(R, X), modified(R), absAp(R), omittedAtomFrom(X1, R).$$

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., not the cause of inconsistency of $\widehat{I}$), we add the constraint $\leftarrow ab_l(X), not\ someFaulty$.

For an abstract answer set $\widehat{I}$, we denote by $\mathcal{T}[\Pi, \widehat{I}]$ the program $\mathcal{T}_{meta} \cup \mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}] \cup \mathcal{T}_{badomit} \cup Q^{\overline{A}}_{\widehat{I}}$.

Example 4. For the following program Π, $\widehat{I} = \{b\}$ is a spurious answer set of the abstraction for $A = \{a, d\}$:

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

$\mathcal{T}[\Pi, \widehat{I}]$ 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 5. 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	$\widehat{\Pi}_2$
$r1 : a \leftarrow\ b.$		$r1 : a \leftarrow b.$	
$r2 : b \leftarrow not\ c, a.$	$\{b\} \leftarrow not\ c.$	$r2 : b \leftarrow not\ a, c.$	
		$r3 : c.$	$c.$

$AS(\Pi_1) = \{\}$ and omitting a creates a spurious answer set $\{b\}$ disregarding that b in unfounded. $\mathcal{T}[\Pi_1, \{b\}]$ gives $inPosLoop(b, a), ap(r1), ap(r2), ab_l(b), badomit(a)$

The program Π_2 is unsatisfiable, due to the odd loop of a and b. When both atoms are omitted, this odd loop is disregarded, which causes a spurious answer set $\{c\}$. $\mathcal{T}[\Pi_2, \{c\}]$ gives $ap(r3), inOddLoop(b, a), ab_l(b), ap(r1), bl(r2), badomit(a)$.

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

Proposition 13. *If $\widehat{I}$ is spurious, then for every answer set $S \in AS(\mathcal{T}[\Pi, \widehat{I}])$, $badomit(\alpha) \in S$ for some $\alpha \in A$.*

Proof (sketch). For a spurious $\widehat{I}$, $\Pi \cup Q^{\overline{A}}_{\widehat{I}}$ causes an inconsistency, which can either be due to (i) an unsatisfied rule, (ii) an unsupported atom, or (iii) unfoundedness or an involvement in an odd loop, all introduced due to forcing the answer set to match $\widehat{I}$. $\mathcal{T}[\Pi, \widehat{I}]$ catches all possibilities. □

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

Corollary 14. *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^{rule}_{Π}, shows the positive/negative dependencies similarly as in G_{Π}, but at a rule-level, where the vertices are rules $r \in \Pi$. For a set A of atoms, n_A denotes the maximum length of a (non-cyclic) path in G^{rule}_{Π} 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 15. *Given a program Π, a set A of atoms, and a spurious $\widehat{I} \in AS(omit(\Pi, A))$, after at most n_A iterations of finding a bad omission with $\mathcal{T}[\Pi, \widehat{I}]$ and refinement, no abstract answer set matching $\widehat{I}$ 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 where $H(r_i) \cap A \neq \emptyset, 0 \leq i \leq n_A$ and $B(r_i) \cap A \neq \emptyset, 0 \leq i < n_A$. (In case of more than one such dependency paths with several rules causing inconsistencies, the returned set of $badomit$s from $\mathcal{T}[\Pi, \widehat{I}]$ allows one to refine the rules in parallel). Let $\widehat{I} \in AS(omit(\Pi, A))$, assume r_0 has spurious behavior w.r.t. $\widehat{I}$, and, wlog, $\widehat{I} \models B(r_i) \setminus A$ for all $i \leq n_A$.

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

Recall that Proposition 5 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.

Figure 2: Results (with upper limit on $badomit$ # per step)

Π	$\frac{\lvert A_{init}\rvert}{\lvert\mathcal{A}\rvert}$	$\frac{\lvert A_{final}\rvert}{\lvert\mathcal{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert \subset_{\min} \rvert}{\lvert\mathcal{A}\rvert}$	t (sec)
GC	0.49	0.36	1.1	1.33	0.12	4.42
	0.74	0.20	2.6	2.73	0.12	5.78
	1.00	0.16	3.2	3.36	0.13	5.97
	top-down				0.13	6.86
AA	0.50	0.17	3.1	5.52	0.17	40.58
	0.75	0.18	2.9	5.76	0.16	40.35
	1.00	0.00	2.0	3.33	0.19	42.60
	top-down				0.19	42.19
SC	0.49	0.49	0.0	0.59	0.11	0.91
	0.74	0.38	0.7	1.17	0.12	1.13
	1.00	0.43	1.0	1.73	0.14	1.16
	top-down				0.14	1.69

Evaluation on Unsatisfiable Problems

The aim of these evaluations is to observe the use of abstraction and refinement for achieving an over-approximation of a program that is still unsatisfiable and to compute the $\subset$-minimal blockers of the programs, which projects away the part that is unnecessary for the unsatisfiability.

For finding $\subset$-minimal blocker sets, we additionally compare the top-down method to the bottom-up method. The top-down method is by checking if omitting an atom from Π preserves unsatisfiability, if yes, it is added to A. The bottom-up method initially omits a certain percentage of the atoms and refines the abstraction until an unsatisfiable abstract program is reached. Then, a search for $\subset$-minimal blocker sets is done similarly, with the remaining atoms.

Benchmarks. We consider three benchmark problems, two of which are based on graphs, and focus on the unsatisfiable instances.

Graph Coloring (GC). We obtained the generator for the graph coloring problem[1] submitted to ASPCOMP 2013 (Alviano et al. 2013). We generated 30 graph instances with nodes size varying from 25 to 50, which were mostly 3-colorable. Thus, in order to have unsatisfiability, we asked for 2-colorability of the instances.

Abstract Argumentation (AA). Abstract argumentation frameworks are based on graphs to represent and reason about arguments. They have a broad set 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) (Watts and Strogatz 1998) instances that are generated by (Cerutti, Giacomin, and Vallati 2016) and are unsatisfiable for existence of stable [2] extensions. We focused on the instances with 100 arguments (i.e., nodes) where each argument is connected (i.e., has an edge) to its 6 nearest neighbor and it is connected to the remaining arguments with a probability $\beta\%$, $\beta \in \{10, 30, 50, 70, 90\}$.

[1] www.mat.unical.it/aspcomp2013/GraphColouring

[2] www.dbai.tuwien.ac.at/research/project/argumentation/systempage/Data/stable.dl

Figure 3: Additional results with $badomit$ # minimization

Π	$\frac{\lvert A_{init}\rvert}{\lvert\mathcal{A}\rvert}$	$\frac{\lvert A_{final}\rvert}{\lvert\mathcal{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert \subset_{\min} \rvert}{\lvert\mathcal{A}\rvert}$	t (sec)
AA	0.50	0.27	6.4	11.22	0.17	36.58
	0.75	0.41	8.7	17.23	0.15	31.84
	1.00	0.59	19.6	50.00	0.13	22.72

Strategic Companies (SC). As a non-graph problem, we considered the strategic companies problem with the encoding and simple instances provided in (Eiter et al. 1998). In order to have unsatisfiability, we added an additional constraint to the encoding, which forbids having all of the companies that produces a product to be strategic. SC is an example of canonically disjunctive programs. The rules with disjunctive heads, e.g., $a \vee b \leftarrow c$, can be split into choice rules $\{a\} \leftarrow c; \{b\} \leftarrow c$ at the cost of introducing spurious guesses. This program with the split can be seen as an over-approximation of the original program. The above method can then be applied similarly to such programs.

Preliminary results. The results are shown in Figure 2. The tests[3] were run on a Intel Core i5-6200U CPU 2.30GHz machine using Clingo 5.2, 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, and of the atoms in SC. We show the overall average of 10 runs for each instance.

The first three rows under each category show the bottom-up approach. The columns $|A_{init}|/|\mathcal{A}|$ and $|A_{final}|/|\mathcal{A}|$ 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 the $\subset$-minimal blocker set. For bottom-up approach, search starts from A_{final} and for top-down approach, it starts from $\mathcal{A}$. In each refinement step, the number of determined $badomit$ atoms are minimized to be at most $|A|/2$, while Figure 3 shows its full minimization.

Figure 2 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 not needed (more than 80%) atoms. Additionally, with a bottom-up method it is possible to reach a $\subset$-minimal blocker set which is smaller in size than the ones obtained with the top-down method.The abstraction approach can also be useful if there is a desire to find some (non-minimal) blocker.

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 reach a timeout, e.g., no optimal solution for 45 nodes in GC in 10 min. Figure 3 shows the result of applying minimization in the refinement for the AA instances. As expected, adding the smallest set of $badomit$ atoms back makes it possible to reach a larger omission A_{final} that keeps unsatisfi-

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

ability (e.g., AA with 100% A_{init}: A_{final} is 59% instead of 0% as in Figure 2). On the other hand, this requires to have more refinement steps (Ref #) to reach some unsatisfiable abstract program, which also adds to the overall time.

The $\subset$-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 that each 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.

We remark that our focus in this initial work is on the usefulness of the abstraction approach on ASP, and not on the scalability. Further implementation improvements and optimizations will make it possible to argue on the efficiency.

Discussion

Extensions. Lifting the framework to strong (classical) negation is easily possible. In particular, it can be handled as common, where technically only atoms (positive literals) remain, with neg_α representing negative original literals. We focused in this seminal paper on non-disjunctive programs. Splitting disjunctive programs yields an over-approximation, and applying the abstraction method preserves reasons for inconsistency. The approach can be extended to consider disjunctive programs, by also extending the existing debugging approaches for use in the refinement.

Non-ground case. In non-ground programs, the current method removes all occurrences of a predicate from the program. Lifting this to a more fine-grained abstraction is easily possible, by using auxiliary constraints in the rule body. Basically, if we have an atom $\alpha = p(t_1, ..., t_k)$ that we want to omit and $\alpha' = p(t'_1, ..., t'_k)$ occurs in the body of a rule, then (assuming that α and α' share no variable), we have to add a constraint $\theta(var(\alpha'))$ to the rule body to single out the instances of α' from the ones of α. E.g., $\alpha = p(a, Y)$ and $\alpha' = p(X, Z)$ would lead to the constraint $\theta(X, Z) = X \neq a$. This clearly generalizes the ground case (the formula θ is either $\top$ or $\bot$) and also subsumes the current case of omitting the predicate by having $p(X_1, ..., X_k)$ and $\theta(var(\alpha')) = \bot$.

For determining bad omissions in the non-ground programs, if lifting the current debugging rules is not scalable, other meta-programming ideas (Gebser et al. 2008; Oetsch, Pührer, and Tompits 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 the solving of the debugging more difficult.

Further optimizations. The selection of atoms to omit can be done with further heuristics. For example, in order to determine the set of atoms for initial omission, studying the dependency graph of the program to catch the atoms that cause fewest dependencies may be worthwhile.

Related Work

Although abstraction is a well-known approach to reduce problem complexity, it has not been considered so far in ASP. Atom omission is different from forgetting (Leite 2017), as it aims at an over-approximation of the original program that may not be faithful and does not resort to language extensions such as nested logic programs that otherwise might be necessary. Abstraction has been studied in logic programming (Cousot and Cousot 1992), but stable semantics was not addressed. Pattern databases (Edelkamp 2001) is a similar abstraction notion considered in planning, which is on projecting the state space to a set of variables.

ASP Debugging Investigating inconsistent ASP programs has been addressed with the works on debugging (Brain et al. 2007; Oetsch, Pührer, and Tompits 2010; Dodaro et al. 2015; Gebser et al. 2008), where the basic assumption is having an inconsistent program and a candidate solution that is expected to hold. 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, but the aim is not to debug the program itself, but to debug (and refine) the abstraction that is constructed.

Different from other works, (Dodaro et al. 2015) computes 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 found set of blocker atoms 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 are of mutually incompatible constraints. However for large programs, the smallest input program where the error happens needs to be manually found. Another related work (Pontelli, Son, and Elkhatib 2009) is on giving justifications for the truth values of atoms w.r.t. an answer set with a graph that encodes the reasons.

Unsatisfiable cores in ASP A well-known notion for unsatisfiability is the minimal unsatisfiable subset (MUS) (unsatisfiable cores) (Liffiton and Sakallah 2008; Lynce and Silva 2004). It is based on computing a minimal subset of constraints that explains why a given system of constraints is unsatisfiable.

In the ASP context, the notion of cores has been used for computing optimal answer sets (Alviano and Dodaro 2016; Andres et al. 2012), where for a given consistent program, cores are used as an underestimates for the cost of the optimal answer set. However, if the program is inconsistent, such a core is not obtained. A recent work (Alviano et al. 2018) uses unsatisfiable core computation for cautious reasoning.

Relation to spurious answer sets. An *unsatisfiable (u-) core* is an assignment I_C over $C \subseteq \mathcal{A}$ such that Π has no answer set I' compatible with I_C, i.e., $I' = I_C$. Spurious answer sets and unsatisfiable cores are related as follows.

Proposition 16. *If I is a spurious answer set of $omit(\Pi, A)$, then I is a u-core of Π; furthermore, if A is largest, then I is a minimal core.*

However, minimal cores C are not necessarily spurious answer sets of the corresponding omission $A = \mathcal{A} \setminus C$. E.g.,

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

has the minimal core $C{=}\{b\}$; but $C{\notin}AS(omit(\{r\}, \{a\}))$. Intuitively, the reason is that C lacks foundedness for the abstraction. Thus, spurious answer sets are a more fine-grained notion of relative inconsistency than u-cores.

Conclusion

In this paper, we have introduced a novel approach for abstracting ASP programs by omitting atoms and constructing over-approximations. The refinement of the abstraction for the unavoidably introduced spurious answer sets is based on debugging the spuriousness through the badly omitted atoms. This approach can be used for different purposes. We demonstrate the use in obtaining a representation of projection of a program and catching the strong cause of inconsistency with obtaining blockers for answer set existence.

Outlook. This first work on abstraction for ASP has potential for enhancement and further developments. The abstraction method can be made more sophisticated to avoid introducing too many spurious answer sets. Better results can be expected with subtle improvements on the implementation, while even in the non-optimized way it is still possible to obtain usable results.

Acknowledgements

This work has been supported by the Austrian Science Fund (FWF) project W1255-N23. We thank the anonymous reviewers for their valuable feedback.

References

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.; 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*, 42–53. Springer.

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*, volume 17, 211–221. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Brain, M.; Gebser, M.; Pührer, J.; Schaub, T.; Tompits, H.; and Woltran, S. 2007. Debugging asp programs by means of asp. In *Proc. LPNMR*, 31–43. Springer.

Buss, S.; Krajìček, J.; and Takeuti, G. 1993. On provably total functions in bounded arithmetic theories. In Clote, P., and Krajìček, J., eds., *Arithmetic, Proof Theory and Computational Complexity*. Oxford University Press. 116–61.

Cerutti, F.; Giacomin, M.; and Vallati, M. 2016. Generating structured argumentation frameworks: AFBenchGen2. In Baroni, P.; Gordon, T. F.; Scheffler, T.; and Stede, M., eds., *Proc. COMMA*, volume 287, 467–468. IOS Press.

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.

Cousot, P., and Cousot, R. 1992. Abstract interpretation and application to logic programs. *The Journal of Logic Programming* 13(2):103 – 179.

Dodaro, C.; Gasteiger, P.; Musitsch, B.; Ricca, F.; and Shchekotykhin, K. 2015. Interactive debugging of non-ground asp programs. In *Proc. LPNMR*, 279–293. Springer.

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

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.

Faber, W.; Leone, N.; and Pfeifer, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proc. JELIA*, volume 3229 of *LNCS*, 200–212. Springer.

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.; Pührer, J.; Schaub, T.; and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proc. AAAI*, volume 8, 448–453.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9(3):365–385.

Janota, M., and Marques-Silva, J. 2016. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell.* 233:73–83.

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

Liffiton, M. H., and Sakallah, K. A. 2008. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning* 40(1):1–33.

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.

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.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

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

Watts, D. J., and Strogatz, S. H. 1998. Collective dynamics of "small-world" networks. *Nature* 393:440–442.