On Interactively Establishing Fact-Independent Consistency in Answer Set Programs

Andre Thevapalan[0000−0001−5679−6931] and Gabriele Kern-Isberner[0000−0001−8689−5391]

Technische Universität Dortmund, Dortmund, Germany
{andre.thevapalan,gabriele.kern-isberner}@tu-dortmund.de

1 Introduction

Using logic programs in real-world applications often requires that a program is usable with different sets of input data. Even by limiting the set of valid inputs, it is not always guaranteed that the program is free of any conflicting information that causes contradictions. The resolution of conflicts in logic programs normally requires domain-specific expert knowledge if one wants to find the professionally and cognitively most adequate solution. Thus, an expert is required who is able to modify rules such that the program becomes free of possible contradictions and the represented knowledge remains compliant with professional knowledge and standards. We propose a general framework that computes suitable solutions for resolving potential conflicts and that involves the expert in the conflict resolution process to obtain the most suitable solution(s).

2 Extended Logic Programs

In this paper, we look at non-disjunctive *extended logic programs* (ELPs) [3] over a set $\mathcal{A}$ of propositional atoms. A classical literal L is either an atom A or a negated atom $\neg A$. For a literal L, the *strongly complementary* literal $\overline{L}$ is $\neg A$ if $L = A$ and A otherwise. For a set S of classical literals, $\overline{S} = \{\overline{L} \mid L \in S\}$ is the set of corresponding strongly complementary literals. Then, $Lit_{\mathcal{A}}$ denotes the set $\mathcal{A} \cup \overline{\mathcal{A}}$ of all classical literals over $\mathcal{A}$. A *default-negated literal* L, called *default literal*, is written as $\sim L$. By an *extended literal* L^*, we either mean a (classical) literal L or a default-negated literal $\sim L$. The set of all extended literals over a set of atoms $\mathcal{A}$ will be denoted by $Lit^*_{\mathcal{A}}$, i. e., $Lit^*_{\mathcal{A}} = Lit_{\mathcal{A}} \cup \sim Lit_{\mathcal{A}}$ where $\sim Lit_{\mathcal{A}} = \{\sim A \mid A \in Lit_{\mathcal{A}}\}$. A classical literal L is true in S iff $L \in S$ and a default literal $\sim L$ is true in S iff $L \notin S$. A set X of extended literals is true in S (symbolically $S \vDash X$) iff every extended literal $L^* \in X$ is true in S. Two extended literals L^*,K^* are *atom-related* if L^*,K^* are based on the same atom.

We are now ready to specify the form of ELPs.

A *rule* r is of the form $L_0 \leftarrow L_1, \dots, L_m, \sim L_{m+1}, \dots, \sim L_n.$, with classical literals $L_0, \dots, L_n$ and $0 \leq m \leq n$. The literal L_0 is the *head* of r, denoted by $H(r)$, and $\{L_1, \dots L_m, \sim L_{m+1}, \dots \sim L_n\}$ is the *body* of r, denoted by $B(r)$. Furthermore $\{L_1, \dots, L_m\}$ is denoted by $B^+(r)$ and $\{L_{m+1}, \dots, L_n\}$ by $B^-(r)$. A

rule r with $B(r) = \emptyset$ is called a *fact*, and r is called a *constraint* if it has an empty head. For a fact $L.$, we will call the corresponding literal L a *fact literal*. A set $\mathcal{F}$ of facts is *consistent* if the set of fact literals in $\mathcal{F}$ is consistent. Atoms that occur in the rule bodies of $\mathcal{P}$ but not in any rule head will be called *extensional atoms*. We will denote the set of extensional atoms by $\mathcal{E}_{\mathcal{P}}$, and the set of classical literals over $\mathcal{E}_{\mathcal{P}}$ will be denoted by $Lit\mathcal{E}_{\mathcal{P}}$. A rule r will be called a *complex rule* if r is neither a fact nor a constraint. An *extended logic program (ELP)* $\mathcal{P}$ is a set of rules and a rule $r \in \mathcal{P}$ is *applicable* iff there is a consistent set S of classical literals such that $S \vDash B(r)$. For the rest of this paper, we will assume that every rule in a given logic program is applicable. Given a set S of classical literals, a rule r is true in S (symbolically $S \vDash r$) iff $S \vDash H(r)$ holds whenever $S \vDash B(r)$. In this case we also say that S *satisfies* r. A rule body $B(r)$ will be called *satisfiable* if there exists a set S of classical literals such that $S \vDash B(r)$. Given an ELP $\mathcal{P}$ without default negation, the *answer set* of $\mathcal{P}$ is either (a) the smallest set $S \subseteq Lit_{\mathcal{A}}$ such that S is consistent and $S \vDash r$ for every rule $r \in \mathcal{P}$, or (b) the set $Lit_{\mathcal{A}}$ of classical literals.

The *reduct* $\mathcal{P}^S$ of a program $\mathcal{P}$ relative to a set S of classical literals is defined by $\mathcal{P}^S = \{H(r) \leftarrow B^+(r). \mid r \in \mathcal{P}, B^-(r) \cap S = \emptyset\}$. In general, an *answer set* of an ELP $\mathcal{P}$ is determined by its reduct: A consistent set S of classical literals is an *answer set* of $\mathcal{P}$ if it is the answer set of $\mathcal{P}^S$ [3]. $\mathcal{P}$ is *consistent* if it has one or more answer sets S such that $S \neq Lit_{\mathcal{A}}$, otherwise $\mathcal{P}$ is *inconsistent*.

We partition an ELP $\mathcal{P}$ into a set $\mathcal{P}_{\mathcal{F}}$ of facts (*input*), and a set $\mathcal{P}_{\mathcal{C}}$ of rules (*program core*). A program core $\mathcal{P}_{\mathcal{C}}$ can only comprise complex rules and an input $\mathcal{P}_{\mathcal{F}}$ for $\mathcal{P}_{\mathcal{C}}$ is *valid* iff $\mathcal{P}_{\mathcal{F}}$ is a consistent set of facts over $Lit\mathcal{E}_{\mathcal{P}}$. The set of all valid inputs $\mathcal{P}_{\mathcal{F}}$ for a program core $\mathcal{P}_{\mathcal{C}}$ will be denoted by $I(\mathcal{P}_{\mathcal{C}})$. By $\Pi(\mathcal{P}_{\mathcal{C}}) = \{\mathcal{P}_{\mathcal{C}} \cup \mathcal{P}_{\mathcal{F}} \mid \mathcal{P}_{\mathcal{F}} \in I(\mathcal{P}_{\mathcal{C}})\}$, we denote the set of all programs $\mathcal{P}_{\mathcal{C}}$ extended by a valid input $\mathcal{P}_{\mathcal{F}}$ for $\mathcal{P}_{\mathcal{C}}$.

We will distinguish between *contradictory* and *incoherent* programs. An inconsistent program will be regarded as *contradictory* whenever complementary (head) literals are (partly) causing inconsistency. As shown in [6], in all other cases an odd number of negative dependency loops are responsible for the inconsistency. These types of inconsistent programs will be called *incoherent*. In the rest of this paper, we assume that any given logic program is coherent, i. e., free of odd negative dependency loops.

In order to present a method that modifies a program $\mathcal{P}_{\mathcal{C}}$ such that any $\mathcal{P} \in \Pi(\mathcal{P}_{\mathcal{C}})$ is not contradictory, we introduce the notion of *uniformly non-contradictory program cores*.

Definition 1 (Uniformly Non-Contradictory Program Core). *A program core $\mathcal{P}_{\mathcal{C}}$ of an ELP $\mathcal{P}$ over $\mathcal{A}$ is* uniformly non-contradictory *if for every valid input $\mathcal{P}_{\mathcal{F}}$ for $\mathcal{P}_{\mathcal{C}}$, $\mathcal{P}_{\mathcal{C}} \cup \mathcal{P}_{\mathcal{F}}$ is not contradictory.*

We will now define how $\mathcal{P}_{\mathcal{C}}$ can be modified such that it becomes uniformly non-contradictory and how the user can be involved in the process to find the most adequate modification for $\mathcal{P}_{\mathcal{C}}$.

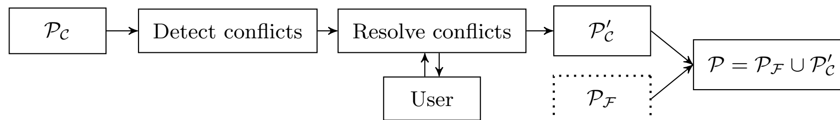

Fig. 1. Framework Overview

3 Interactive Conflict Resolution Process

The goal of the conflict resolution process is to ensure that given a program core $\mathcal{P}_{\mathcal{C}}$, every ELP $\mathcal{P} \in \Pi(\mathcal{P}_{\mathcal{C}})$ is non-contradictory. As illustrated in Figure 1, to that end, we first *identify* what may cause contradictions, and we will then show how to find the causes (*conflict detection*) and remove them (*conflict resolution*). Subsequently, we will briefly discuss how the results can be used to integrate the expert into the conflict resolution process.

3.1 Conflicts

To find rules that could potentially lead to contradictions, one has to look at rules with strongly complementary head literals.

Definition 2 (Conflicting Rules, Conflict [8]). *Suppose an ELP $\mathcal{P}$. Two rules $r, r' \in \mathcal{P}_{\mathcal{C}}$, $r \neq r'$, are* conflicting *(written as $r \bowtie r'$) if $H(r)$ and $H(r')$ are strongly complementary and there exists a consistent set of classical literals $S \subseteq Lit_{\mathcal{A}}$ such that $S \models B(r) \cup B(r')$. A* conflict *is a pair (r, r') of rules such that r, r' are conflicting. We will denote the set of all conflicts (r, r') in an ELP $\mathcal{P}$ by $Conflicts(\mathcal{P})$, and correspondingly the set of all conflicts (r, r') involving a rule r will be denoted by $conflicts(r)$. Furthermore, we will refer to the set of rules $r' \in \mathcal{P}$ that are conflicting with r in $\mathcal{P}$ as* adversarial rules *of r, denoted by $Adv(r)$, i. e., $Adv(r) = \{r' \mid r \bowtie r', r' \in \mathcal{P}\}$. We will call two rules r, r'* non-conflicting *iff $r \bowtie r'$ does not hold.*

Proposition 1. *Given an ELP $\mathcal{P}$ over $\mathcal{A}$, its program core $\mathcal{P}_{\mathcal{C}}$ is uniformly non-contradictory if $\mathcal{P}_{\mathcal{C}}$ does not contain any conflicts.*

By restricting the set of possible inputs $\mathcal{P}_{\mathcal{F}}$, we also ensure that contradictions via a rule in $\mathcal{P}_{\mathcal{C}}$ and a fact in $\mathcal{P}_{\mathcal{F}}$ cannot arise even if $\mathcal{P}_{\mathcal{C}}$ is conflict-free since those kind of contradictions are not caused by conflicts. Therefore, in the following, $\mathcal{P}_{\mathcal{F}}$ will only consist of fact literals over $\mathcal{E}_{\mathcal{P}}$.

3.2 Conflict Detection

For two rules with complementary heads to be conflicting, both their bodies have to be satisfiable by at least one consistent set of literals.

Theorem 1. *Let $\mathcal{P}$ be a program with rules $r, r' \in \mathcal{P}_{\mathcal{C}}$. Two rules r, r' are conflicting if and only if*

(CP1) *$H(r), H(r')$ are strongly complementary, and*
(CP2) *$B^+(r_1) \cap B^-(r_2) = \emptyset$ such that $r_1, r_2 \in \{r, r'\}, r_1 \neq r_2$, and*
(CP3) *$B^+(r) \cup B^+(r')$ is consistent.*

Consequently, a conflict between two rules can be resolved if one or both rule bodies are modified in a way that they contain complementary literals. We can see that r, r' are not conflicting whenever $B(r)$ contains a literal L such that $B(r')$ contains a literal that is either strongly or default-complementary to L.

Definition 3 (Conflict-Preventing Literals). *Given an ELP $\mathcal{P}$ and two rules $r, r' \in \mathcal{P}_{\mathcal{C}}$ with complementary heads, two extended literals $L^* \in B(r)$, $K^* \in B(r')$ are* conflict-preventing *if L^* and K^* are atom-related and complementary.*

Proposition 2. *Let $\mathcal{P}$ be an ELP with two rules $r, r' \in \mathcal{P}$ with complementary heads. The rules r, r' are non-conflicting iff there exist two extended literals $L^* \in B(r), K^* \in B(r)$ such that L^*,K^* are conflict-preventing.*

In order to prevent that two conflicting rules are satisfied, we, therefore, have to ensure that these rules contain conflict-preventing literals.

3.3 Conflict Resolution

We will now outline a method that will produce different possible resolution options for each rule that is in conflict with other rules. The main goal is that each possible solution resolves the conflict without unnecessary changes. In other words, all modifications done to the program in the course of the conflict resolution have to be justified changes in the technical and logical sense. In this approach, we will solely focus on the resolution of conflicts by adding of extended literals to rule bodies which leads to *cautious changes* since it can be shown that these kind of changes will not cause new additional conflicts. Thus, we introduce the notion of λ-*extensions*.

Definition 4. *Given an ELP $\mathcal{P}$ over $\mathcal{A}$ and a rule $r_i \in \mathcal{P}$ with $Adv(r_i) \neq \emptyset$, a* conflict-resolving extension *$\lambda(r_i)$* for *r_i is a (minimal) set of extended literals $L^* \in Lit^*_{\mathcal{A}}$ such that the addition of $\lambda(r_i)$ to the body of r_i, i. e.,*

$$r_i'\colon H(r_i) \leftarrow B(r_i), \lambda(r_i). \tag{1}$$

resolves all conflicts in *conflicts*(r_i) simultaneously*, i. e., replacing r_i with r_i' in $\mathcal{P}$ leads to conflicts$(r_i') = \emptyset$, while the extended rule r_i' remains applicable. A rule r_i extended by a conflict-resolving extension $\lambda(r_i)$ will be called a* λ-extension of r_i.

Given a rule r_i, every λ-extension $\lambda(r_i)$ is a solution that can be suggested to the expert.

To compute all possible conflict-resolving-extensions, the body literals of r_i and its adversaries in $Adv(r_i)$ have to be examined. We now briefly describe the general steps to gather possible λ-extensions.

1. Find all suitable[1] minimal sets β of extended literals such that each β contains exactly one extended literals of every $B(r_j)$, $r_j \in Adv(r_i)$.
2. For every set β, compute all *negated forms* $neg(\beta)$ such that each $neg(\beta)$ contains every extended literal $L^* \in \beta$ in a possible[2] negated form.

It can be shown that given a set β, every such negated set $neg(\beta)$ then contains a conflict-preventing literal for every rule $r_j \in Adv(r_i)$ and is, therefore, a valid λ-extension for r_i.

3.4 User Interactions

It is easy to see that dependending on the program and its conflicts, the number of possible solutions can become quite high. The framework should therefore be able to only provide the user with the information that is really needed and illustrate the possible solutions in a way that does not require the expert to be knowlegdeable in logic programming. We therefore propose to make use of argumentative dialogue techniques [9] (e. g., information seeking dialogues [4]) and graph representations [2]. Using argumentative dialogues enables the framework to essentially acquire the missing pieces of information from the user regarding a conflict in order to incrementally determine which of the computed solutions fits best. In these dialogues, suitable graph representations can then be used as a tool to show the user how certain literals are derived in the program and to illlustrate different dependencies between rules and literals.

4 Conclusion and Future Work

We have shown that conflicts are the causes for potential contradictions in program cores and presented a method to compute suitable solutions. Being able to compute possible solutions to resolve conflicts in a program, allows us to propose a general framework that enables an expert who is not familiar with logic programming to make an informed decision on how to resolve each conflict in the program core by presenting them with the suitable options and gradually obtain the most adequate solution. In contrast to related ASP debugging frameworks, e. g., [1, 5, 7], our approach aims to establish and maintain consistency in program cores and programs that are subject to updates in general.

In future work, we will present a detailed description of finding the λ-extensions for conflicting rules and develop explicit strategies on how to determine the most suitable solutions. Furthermore, we will explore how the presented framework can also be used to establish coherence in a given logic program.

[1] The exact properties of such a minimal set β will be discussed in future work.
[2] Note that there are no strong complements of default literals.

References

1. Dodaro, C., Gasteiger, P., Reale, K., Ricca, F., Schekotihin, K.: Debugging non-ground ASP programs: Technique and graphical tools. Theory Pract. Log. Program. **19**(2), 290–316 (2019)
2. Fandinno, J., Schulz, C.: Answering the "why" in answer set programming - A survey of explanation approaches. Theory Pract. Log. Program. **19**(2), 114–203 (2019)
3. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–386 (1991)
4. McBurney, P., Parsons, S.: Agent ludens: games for agent dialogues. In: Game-Theoretic and Decision-Theoretic Agents (GTDT 2001): Proceedings of the 2001 AAAI Spring Symposium. pp. 70–77 (2001)
5. Oetsch, J., Pührer, J., Tompits, H.: Stepping through an answer-set program. In: Delgrande, J.P., Faber, W. (eds.) Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6645, pp. 134–147. Springer (2011)
6. Schulz, C., Satoh, K., Toni, F.: Characterising and explaining inconsistency in logic programs. In: Calimeri, F., Ianni, G., Truszczynski, M. (eds.) Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9345, pp. 467–479. Springer (2015)
7. Shchekotykhin, K.M.: Interactive query-based debugging of ASP programs. In: Bonet, B., Koenig, S. (eds.) Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA. pp. 1597–1603. AAAI Press (2015)
8. Thevapalan, A., Kern-Isberner, G.: Towards interactive conflict resolution in asp programs. In: Martínez, M.V., Varzinczak, I. (eds.) 18th International Workshop on Non-Monotonic Reasoning, Workshop Notes. pp. 29–36 (September 2020)
9. Walton, D., Krabbe, E.C.W.: Commitment in dialogue: Basic concepts of interpersonal reasoning. Albany, NY, USA: State University of New York Press (1995)